MASTER EN SCIENCES INFORMATIQUES

Réalisation d'un serveur X d'impression

Delperdange, Thierry; Poleur, Michel

Award date:
1994

Facultés Universitaires Notre-Dame de la Paix de Namur
Institut d'Informatique

Réalisation d'un serveur X d'impression

T. Delperdange
M. Poleur

Mémoire de fin d'études présenté par
Thierry Delperdange et Michel Poleur
en vue de l'obtention du diplôme de
Licencié et Maître en informatique

Promoteur: Jacques Berleur

Année académique 1993-94

RESUME

Ce mémoire a pour double objectif de justifier et de décrire l'implémentation d'un serveur X d'impression. Le besoin d'un tel serveur s'est manifesté lors de l'implémentation d'une boîte à outils virtuelle pour différents environnements graphiques, à savoir Windows, Macintosh, Presentation Manager et OSF/Motif. En effet, contrairement aux trois premiers, OSF/Motif, et le système X-Window en général, ne propose pas de service d'impression. Pour résoudre ce problème, deux stratégies sont possibles: soit les applications prennent totalement en charge l'impression, soit elles délèguent leurs impressions de documents à un programme spécialisé, un serveur d'impression. Cette dernière stratégie, intéressante car présentant des similitudes avec l'architecture du système X-Window, est développée dans ce mémoire.

ABSTRACT

This thesis has a twofold goal: to justify and to describe the implementation of an X print server. The need for such a server appeared during the development of a virtual toolkit for several graphical environments, i.e. Windows, Macintosh, Presentation Manager and OSF/Motif. Indeed, unlike the first three, OSF/Motif, and more generally the X-Window system, does not provide printing services. In order to solve this problem, two strategies are possible: first, the applications must deal with the entire problem of printing themselves, second, they delegate printing to a specialized program, a print server. Because it fits well with the overall philosophy of the X-Window system, the latter is developed in this thesis.

Nous tenons à remercier toutes les personnes qui nous ont aidés, de près ou de loin, à la rédaction de ce mémoire. En particulier, notre promoteur le Père Berleur, pour ses nombreux conseils, Monsieur Mousel et la sympathique équipe du CRP-CU qui nous ont épaulé durant notre stage, l'ensemble des personnes ayant contribué à la réalisation du serveur X d'impression et toute la bande des *maileurs* fous. Cette liste est bien sûr non exhaustive et son ordre n'a pas d'importance.

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION

Dans les environnements graphiques les plus connus tels Microsoft Windows, IBM OS/2 Presentation Manager, Apple Macintosh, les services d'impression et d'affichage sont offerts par le système et sont donc intimement liés. Les fonctions graphiques de ces environnements sont utilisées indifféremment pour adresser l'écran et l'imprimante. Par contre, dans l'environnement graphique X, ces services sont indépendants puisque X n'offre pas de facilités d'impression. C'est pourquoi les développeurs sont contraints de gérer eux-mêmes l'impression au sein de leurs applications.

Pour imprimer sous X, deux stratégies sont possibles. La première, toute application prend totalement en charge l'impression. La seconde, les applications délèguent leurs impressions de documents à un programme spécialisé. Cette dernière stratégie est intéressante car elle présente des similitudes avec l'architecture du système X-Window, programme spécialisé dans l'affichage.

La finalité de ce mémoire est, d'une part, de décrire le raisonnement suivi pour aboutir au concept de système X d'impression et, d'autre part, de montrer la faisabilité d'un tel système.

Lorsqu'une application délègue un service à une application spécialisée, une communication s'établit, nécessitant la présence d'un protocole. Celui-ci permet l'envoi de la demande de service et de la réponse à cette demande. Par analogie au monde réel, on nomme alors *client* l'application demandeuse de service et *serveur* l'application y répondant. Cette définition correspond à celle du modèle Client-Serveur, discutée dans le Chapitre 1. Un système où les applications délèguent leurs impressions à une application spécialisée respecte donc le modèle Client-Serveur.

Après une définition du modèle, le Chapitre 1 expose les types de protocoles et d'interfaces permettant le dialogue entre client et serveur. Ensuite, sur base du modèle de structuration d'une application, les services que celle-ci peut déporter sont dégagés, à savoir, des services de gestion de données, d'exécution de procédures et de présentation. C'est dans la nature de ces services que l'on peut faire la distinction entre les trois types de Client-Serveur, le Client-Serveur de données, le Client-Serveur de procédures et le Client-Serveur de présentation.

Parmi les applications du modèle Client-Serveur de présentation, on trouve le système X-Window, discuté dans le Chapitre 2. Ce système multi-fenêtré multi-plateformes repose sur le protocole X qui régit le dialogue entre une application cliente et un serveur d'affichage. Le respect de ce protocole garantit aux applications clientes un affichage sur tout type de périphérique piloté par un serveur X. Toutefois, il existe une autre application du modèle Client-Serveur de

présentation si on accepte l'analogie entre l'affichage et l'impression. Dans ce cas, le protocole X peut être utilisé pour régir le dialogue entre une application cliente et un serveur d'impression. Un système où les applications délèguent leurs impressions à une application spécialisée respecte donc le modèle Client-Serveur de présentation.

Le Chapitre 2 est consacré à la présentation des caractéristiques du serveur X d'affichage. Si on accepte l'analogie entre l'affichage et l'impression, ces caractéristiques s'appliquent également à un serveur X d'impression. Après avoir défini X et exposé son architecture, une comparaison avec d'autres systèmes d'affichage permettra de mettre en évidence ses principaux avantages. Ensuite, seront abordés les objets manipulés par un serveur, les boîtes à outils de l'interface de programmation disponibles pour le développement des applications clientes, ainsi que le protocole X. La présentation du mécanisme utilisé par les développeurs pour ajouter de nouvelles fonctionnalités au serveur clôture le chapitre.

L'absence de facilités d'impression sous X a posé de sérieux problèmes à la société SILIS et au Centre de Recherche Public - Centre Universitaire de Luxembourg (CRP-CU), partenaires dans le cadre du projet PAGE (Portability of Applications in multiple Graphical Environments). L'objet de ce projet, discuté dans le Chapitre 3, est de réaliser une boîte à outils virtuelle (GRAVITI - GRAphical VIrtual ToolkIt) présentant aux applications une interface de programmation unique quel que soit l'environnement graphique visé. Le module d'impression de GRAVITI dans l'environnement X n'a pu être implémenté étant donné le manque de service d'impression. Les concepteurs de GRAVITI ont choisi un serveur X d'impression pour remédier à ce problème.

Le Chapitre 3 aborde les problèmes de portabilité des applications dans des environnements graphiques multiples. Plusieurs solutions sont discutées, notamment la boîte à outils virtuelle GRAVITI. Le dernier point est consacré à Interscript, une illustration des possibilités de GRAVITI.

Dans des environnements graphiques tels Apple Macintosh et Microsoft Windows, l'analogie entre l'affichage et l'impression est effective puisque les programmeurs utilisent les mêmes fonctions pour adresser l'écran et l'imprimante. La solution du serveur X d'impression confère cette propriété au système X-Window.

Le quatrième chapitre décrit les procédures suivies par les applications Macintosh et Windows pour imprimer un document. Grâce au serveur X d'impression, une application X peut suivre une procédure similaire.

Cinq illustrations de la stratégie selon laquelle toute application cliente X prend totalement en charge ses impressions sont proposées dans le Chapitre 5. Leurs descriptions révèlent des inconvénients, le plus important étant la pénalisation de l'application cliente X en temps et en taille. Ce désavantage est absent dans le cas du serveur X d'impression.

Le Chapitre 5 aborde ainsi la copie écran, la génération de rapports au sein de l'application, la librairie de fonctions d'impression, la librairie de fonctions d'affichage et d'impression et la librairie d'impression d'écran.

Le raisonnement suivi jusqu'à présent tend à montrer que le serveur d'impression est la solution la mieux adaptée à l'environnement X. La faisabilité d'un tel système a été établie par la réalisation d'un serveur X d'impression et d'une application de test. Les détails de cette implémentation font l'objet du Chapitre 6.

Ce chapitre débute par l'exposé de la démarche adoptée, suivi de la présentation de PostScript, un langage de description de page largement répandu et utilisé par le serveur X d'impression. La suite du chapitre est consacrée à la présentation des différentes étapes qui ont conduit à un serveur X d'impression et une application de test opérationnels.

CHAPITRE 1. LE MODÈLE CLIENT-SERVEUR

1.1. Définition

Il nous semble important de préciser, dès l'abord, que nous allons parler du modèle Client-Serveur, et non pas d'architecture Client-Serveur. En effet, ces deux termes sont utilisés indifféremment par les constructeurs et les éditeurs pour qualifier leurs produits ou leurs stratégies. Dans le but d'éviter toute confusion, rappelons que l'on doit parler de modèle Client-Serveur puisqu'il décrit le mode de fonctionnement des applications et que l'on parlera d'un système informatique ayant une architecture de type Client-Serveur lorsqu'il respectera un certain nombre de principes contenus dans le modèle.

Cette précision étant apportée, intéressons-nous au modèle Client-Serveur, et essayons d'en donner une définition générale. On peut dire que ce modèle se résume en un dialogue entre deux programmes. Ce dialogue est établi pour permettre un échange qui peut revêtir différentes formes. On pourra, par exemple, échanger des données ou des informations résultant d'un traitement particulier.

Comme son nom l'indique, le modèle Client-Serveur est composé de deux éléments principaux : le client et le serveur. Le client est le programme qui initie le dialogue, tandis que le serveur se contente de répondre aux demandes du client. Le client formule ses questions sous forme de requêtes. Celles-ci sont envoyées au serveur qui les interprète, puis qui exécute le traitement approprié. Le résultat de cette exécution est ensuite envoyé au client. On parlera alors de service rendu au client par le serveur.

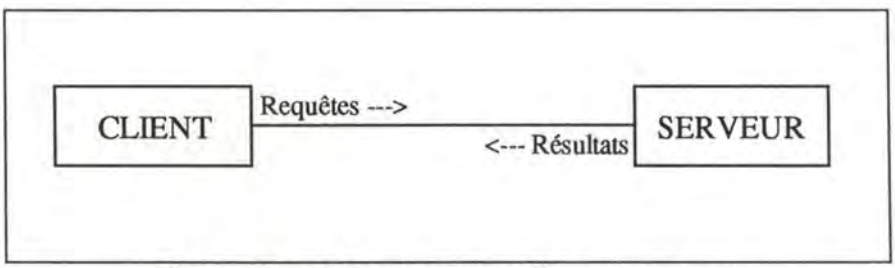

Figure 1 - Dialogue entre client et serveur

Le client est une application, c'est-à-dire un programme spécifique qui exécute une tâche donnée. A l'opposé, se trouve le serveur qui est, quant à lui, un programme générique. En effet, sa tâche est de répondre aux demandes de tous les clients, et cela, quel que soit le client. De plus, il est totalement indépendant des applications clientes. Lorsqu'une application lui envoie une requête, les seules données concernant cette application dont il dispose sont l'adresse réseau du client et le niveau d'autorisation de ce client.

D'une façon plus précise, on peut dire que le modèle Client-Serveur permet à une application (le client) de faire appel à des services extérieurs situés du côté serveur. En effet, au cours de son exécution, une application fait appel à un grand nombre de services. En guise d'exemple, on peut citer les services offerts par le système d'exploitation pour la gestion des fichiers. Il se charge de mener à bien toutes les opérations de lecture et d'écriture sur les différents périphériques, et permet ainsi aux applications de se concentrer uniquement sur la tâche qui leur est demandée. Mais d'autres services plus complexes, ne pouvant être pris en compte par le système d'exploitation, peuvent, eux aussi, être déportés. On pense notamment à la gestion d'une base de données. Ces types de traitement consomment beaucoup de ressources machines et il est donc préférable, dans un souci de performance et d'efficacité, de les faire résider chacun sur un ordinateur dédié appelé serveur.

A ce stade, on est en droit de se demander quels sont les genres de services qui peuvent être rendus aux applications clientes. Dans le but d'exposer la nature de ces services, partons d'un modèle de structuration d'une application cliente[1] (cf. Figure 2).

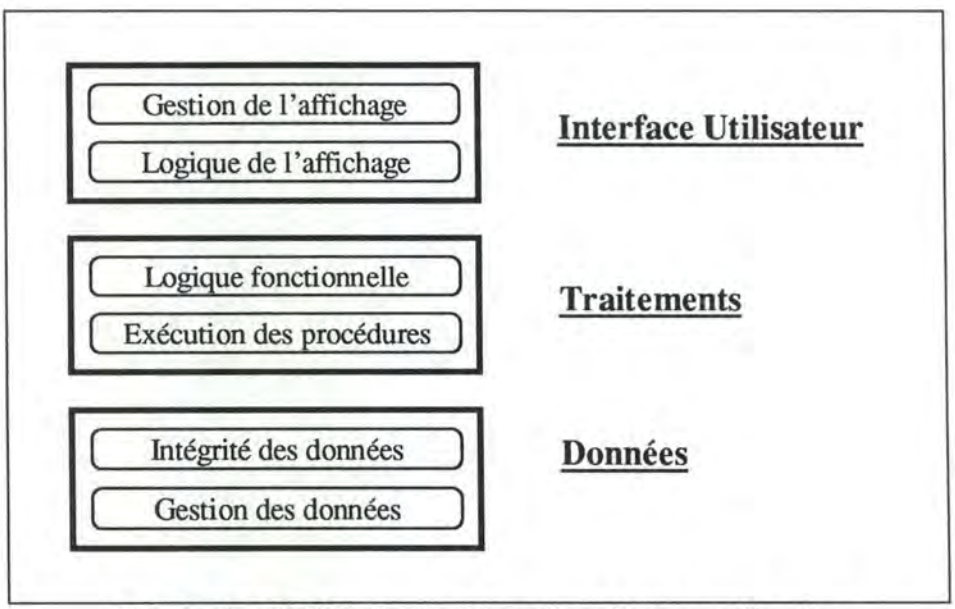

Figure 2 - Modèle de structuration d'une application

[1] LEFEBVRE A., *L'architecture Client-Serveur*, Armand Colin, 1990, p. 32.

Une application est composée de trois parties principales : l'interface utilisateur, les traitements et les données. Remarquons que, déjà à ce niveau, on entrevoit les domaines pour lesquels une application peut faire appel à des services extérieurs. Mais on peut aller plus loin, et subdiviser chacune de ces parties en deux niveaux distincts. On obtient alors une découpe de l'application en six modules :

- La gestion de l'affichage.
  Elle concerne, dans les environnements graphiques, toutes les fonctions de fenêtrage. Elle est prise en charge par l'environnement d'exploitation.

- La logique de l'affichage. Elle transmet à la gestion de l'affichage la description des éléments de présentation.

- La logique fonctionnelle.
  Elle contient l'arborescence algorithmique de l'application.

- L'exécution des procédures.

- L'intégrité des données.
  Elle permet de garantir l'intégrité des données lors de la modification de la base par les procédures.

- La gestion des données.
  Elle recouvre la sélection ou la mise à jour des enregistrements.

Les modules de logique de l'affichage et de logique fonctionnelle sont inséparables, et constituent, de ce fait, le coeur de l'application. Les autres modules sont plus indépendants, et peuvent résider chacun sur un serveur particulier, ils sont ainsi séparés du coeur de l'application. C'est en examinant ces modules, que l'on peut déduire les types de service qu'une application peut déporter. On distingue donc des services de présentation (gestion de l'affichage), d'exécution des procédures, ainsi que de traitements de données. On verra plus tard que la nature des services rendus permet de faire la distinction entre les différents types de Client-Serveur.

1.2. Le dialogue entre client et serveur

Le dialogue entre client et serveur est possible grâce à la présence du réseau. Celui-ci permet l'échange de la demande de service du client vers le serveur, d'une part, et de la réponse du serveur liée à cette demande, d'autre part. Cette communication nécessite un certain nombre de protocoles et d'interfaces de niveau système. Pour fixer les idées, il nous a semblé intéressant de faire un rapprochement avec le célèbre modèle Open Systems Interconnect (OSI) de l'International Standards Organization (ISO) (cf. Figure 3)[2].

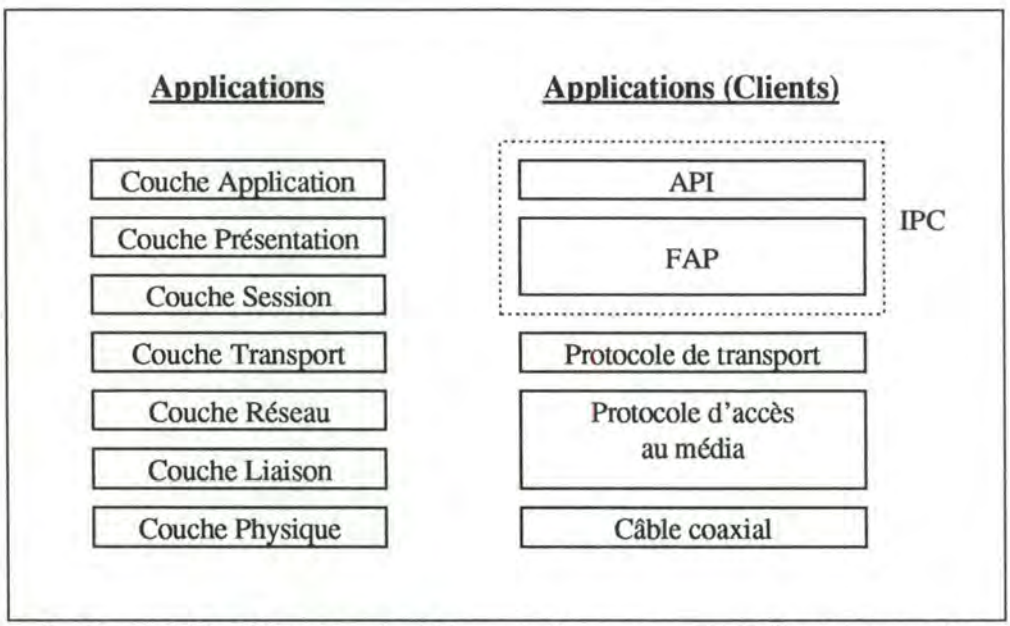

Figure 3 - Le modèle OSI et les implémentations correspondantes pour le modèle Client-Serveur

On fait correspondre à la couche application du modèle OSI l'Application Programming Interface (API) ou interface de programmation au niveau applicatif. Cette interface permet à l'application de bénéficier des services offerts par les serveurs via un certain nombre de fonctions.

Ensuite, on trouve la couche Format and Protocols (FAP) ou protocoles de communication et formats de données, qui regroupe les couches 5 et 6 du modèle OSI. Sa fonction est double, elle assure, grâce à un protocole de communication, la synchronisation du dialogue entre client et serveur, et permet de garantir que le format des données échangées soit identique des deux côtés.

L'API et la FAP constituent à eux deux l'Inter Process Communication (IPC) ou Middleware, c'est-à-dire l'ensemble des couches logicielles qui se trouvent entre l'application et le réseau. Premièrement, ces couches permettent à l'application d'envoyer des requêtes sur le réseau de façon transparente. Ensuite,

[2] ibid., p. 48.

elles s'occupent des conversions éventuelles des protocoles de communication. Et enfin, elles communiquent à l'application les données ou les résultats provenant de sa demande de services. Toutefois, il existe des Middleware plus évolués qui permettent la gestion de paramètres tels que la sécurité ou le recouvrement d'erreurs. En résumé, on peut dire que l'application n'est en contact qu'avec l'API, cette dernière n'est en relation qu'avec la FAP, et la FAP assure la liaison avec les couches réseau.

La première de ces couches est celle de transport. Associé à cette couche, le protocole de transport, qui a pour fonction de rendre les messages émis par l'IPC capables de circuler sur réseau. Pour ce faire, il construit des trames. Celles-ci sont constituées d'une part, du message de l'application, et d'autre part, de renseignements complémentaires indispensables à la circulation de la trame sur le réseau. Une fois constituée, la trame est passée à la couche inférieure : l'accès au média. Le protocole d'accès au média est celui qui gère l'encombrement du réseau. Soit il utilise une technique préventive de collisions entre trames, ou alors une autre méthode de gestion de collisions. Et on trouve enfin la couche physique, qui correspond au média proprement dit, par exemple un câble coaxial.

1.3. Types de modèle Client-Serveur

Le modèle Client-Serveur permet à une application de répartir un certain nombre de services nécessaires à la réalisation de ses objectifs. C'est justement dans la nature de ces services que l'on va trouver le moyen de distinguer les différents types de modèle Client-Serveur.

A ce jour, on reconnaît trois grands types de Client-Serveur, celui de présentation, de données, et de procédures. La Figure 4, inspirée du schéma du Gartner Group[3], illustre bien cette distinction. Sachant que la ligne pointillée représente le réseau, on reconnaît donc là les trois types de Client-Serveur.

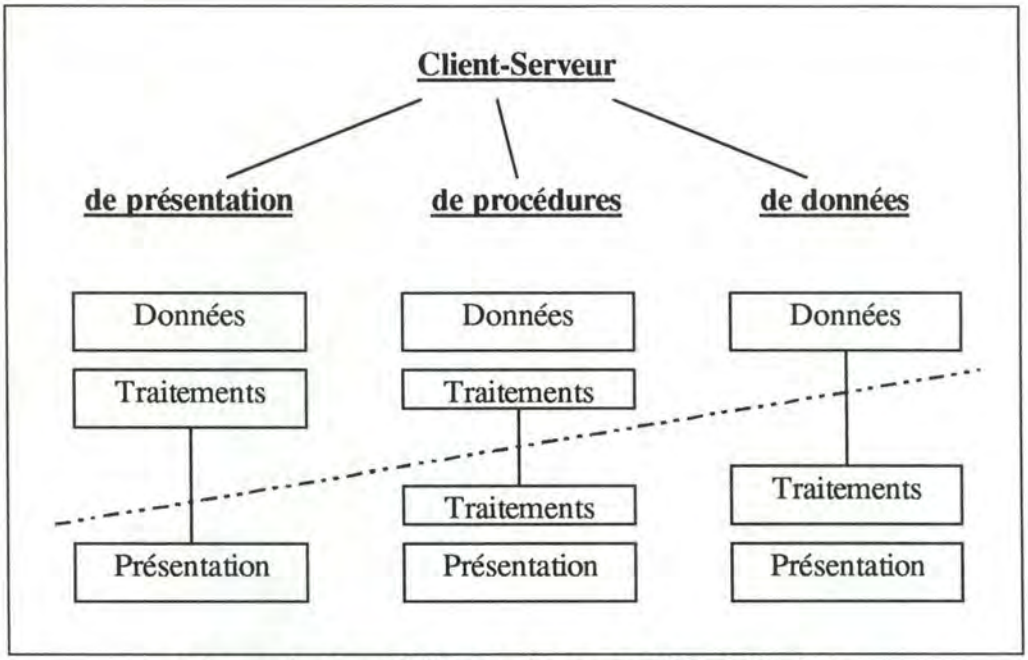

Figure 4 - Les différents types de Client-Serveur

Le premier, celui de présentation, est caractérisé par le fait que l'application fait appel à un service de gestion de l'affichage offert par un système spécialisé. La partie présentation est éloignée du coeur de l'application. Le second est basé sur une répartition des traitements, c'est le Client-Serveur de procédures. Et le dernier, le Client-Serveur de données, met en jeu un serveur dédié qui abrite l'ensemble de la gestion des données de l'application. Cette gestion est donc bien déportée par rapport au coeur de l'application.

[3] ibid., p. 56.

Le Client-Serveur de présentation

Dans le modèle Client-Serveur de présentation, l'application cliente déporte son module de gestion de l'affichage sur un serveur spécialisé, appelé aussi serveur d'affichage. La répartition des tâches entre client et serveur est illustrée par la figure ci-dessous.

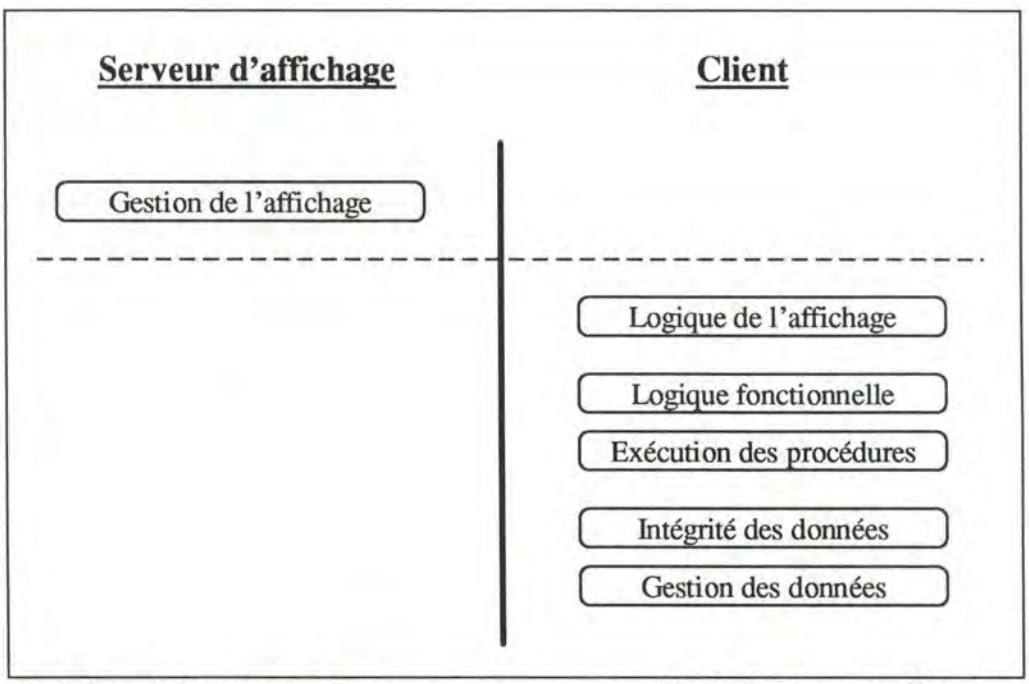

Figure 5 - Répartition des traitements entre serveur d'affichage et application cliente

Dans ce type de Client-Serveur, les applications clientes doivent être capables de s'exécuter à distance et de communiquer avec le serveur d'affichage suivant un protocole bien défini. Puisque le serveur abrite la gestion de l'affichage de l'application cliente, la communication entre client et serveur se résume aux messages échangés entre les deux modules gérant l'interface utilisateur d'une application, à savoir les modules "Logique de l'affichage" et "Gestion de l'affichage". Le dialogue entre ces deux modules est composé, d'une part, des requêtes envoyées par le client décrivant les objets à afficher par le serveur, et d'autre part, par des événements envoyés par le serveur de présentation aux clients, les renseignant sur les actions significatives des utilisateurs.

Un système présentant ces caractéristiques est le système X-Window. Nous y reviendrons assez longuement au chapitre 2.

Le Client-Serveur de données

Le type de Client-Serveur de données est caractérisé par le fait que l'application déporte l'ensemble ou seulement une partie des traitements concernant la gestion des données sur un serveur dédié, appelé serveur de données. Dans la pratique, ce serveur particulier est un Système de Gestion de Base de Données (SGBD).

Au départ, le serveur n'abritait que le module de gestion de données. Aujourd'hui il a évolué et contient l'ensemble des traitements concernant la gestion et l'intégrité des données. Cette évolution est liée aux améliorations des SGBD. Ceux-ci se sont vu dotés de nouveaux mécanismes (contraintes d'intégrité et déclencheurs) permettant de garantir la cohérence des données lors de modifications effectuées par des applications clientes.

Le report du module concernant l'intégrité des données du côté serveur a permis d'apporter une indépendance entre les bases de données et les outils qui les manipulent. En effet, cette méthode autorise n'importe quelle application cliente d'apporter des modifications à des données résidant sur un serveur sans se soucier de préserver l'état de cohérence de la base. Le serveur dispose à cet effet de mécanismes qui sont exécutés indépendamment des applications clientes.

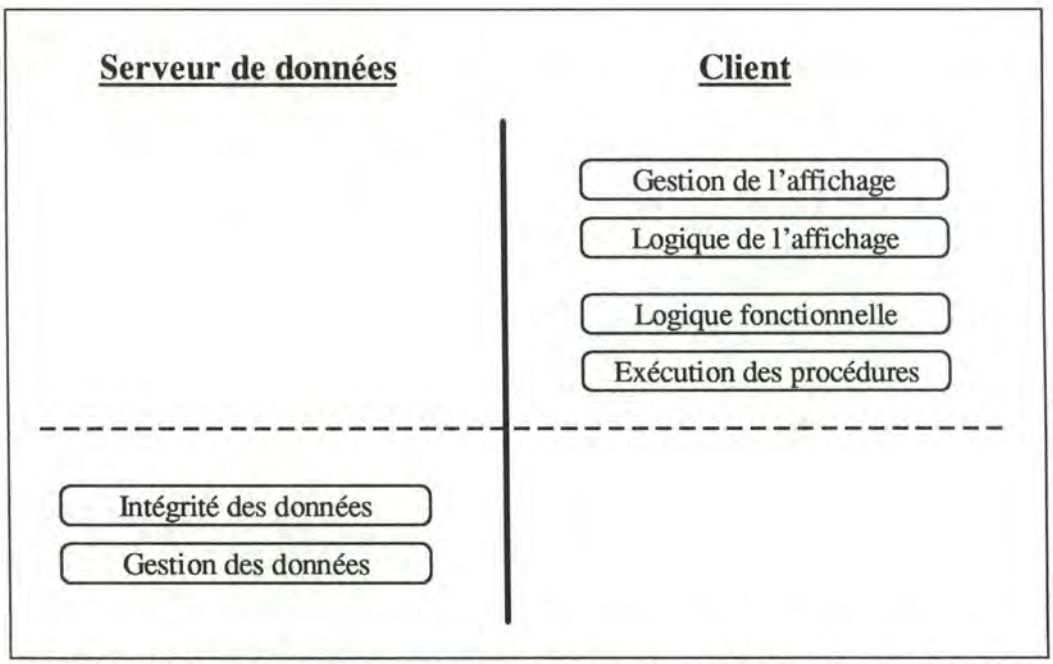

Figure 6 - Répartition des traitements entre un serveur de données évolué et un client

Pour nous permettre de mieux comprendre l'architecture d'un environnement intégrant un serveur de données, d'une part, et le principe de fonctionnement d'un tel système, d'autre part, partons d'un exemple concret.

Soit un environnement constitué de deux machines. La première étant un PC ou une station de travail exécutant une application donnée. La seconde étant le serveur de données, c'est-à-dire une machine dédiée abritant un SGBD. Soit encore un utilisateur travaillant avec l'application. Supposons que cet utilisateur demande à l'application de lui afficher des données se trouvant sur le serveur de données (1). L'application va transformer sa demande en requête, et envoyer cette requête via le réseau au SGBD en utilisant l'IPC (2). Cette requête est réceptionnée du côté serveur et est traitée par le SGBD (3). Un exemple de requête serait de sélectionner les dix meilleures ventes de produits pour une région donnée et cela dans une table de 100000 lignes. Le SGBD va parcourir la table et identifier les dix lignes (4). Ces lignes sont alors envoyées à l'application cliente comme résultat de la requête (5). Il ne reste plus à l'application que de réceptionner le résultat et de l'afficher à l'écran (6).

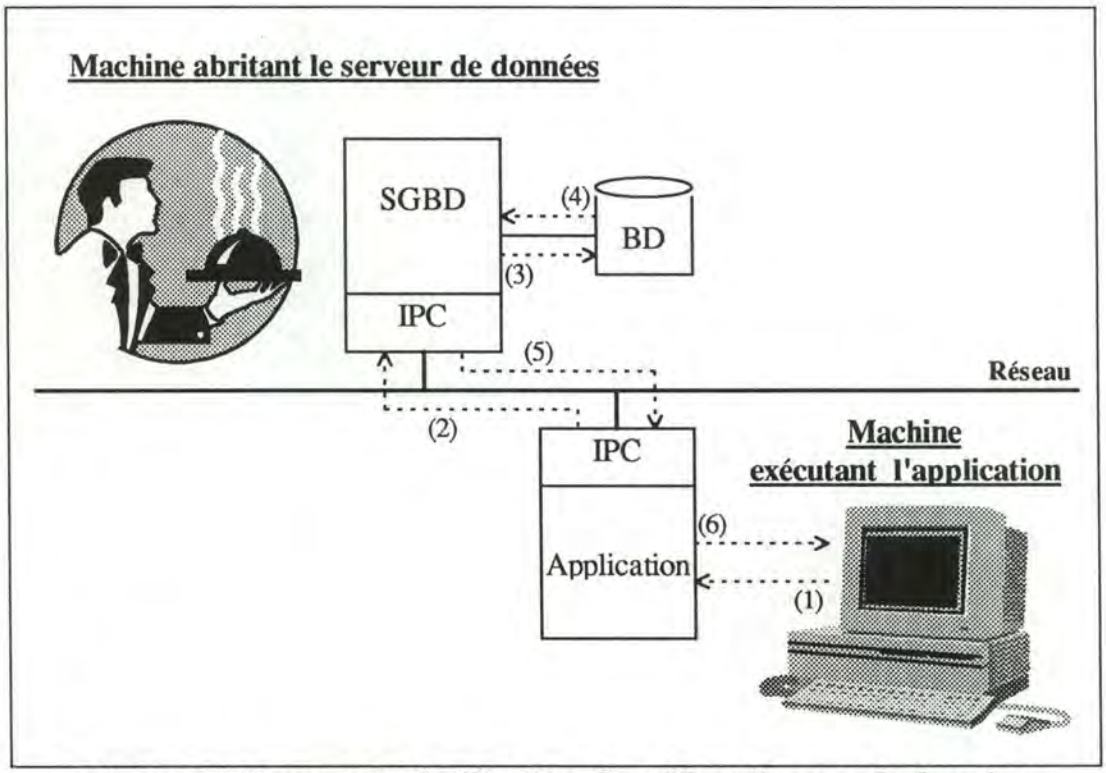

Figure 7 - Un exemple d'utilisation d'un Client-Serveur de données

Un des avantages que procure l'utilisation d'un serveur de données, est la diminution du trafic sur le réseau. Cela est dû au fait de la répartition des tâches entre client et serveur. En effet, toutes les opérations concernant les données sont effectuées du côté serveur, ce qui évite de devoir transférer l'ensemble des informations du côté client. L'application cliente ne réceptionne que les résultats de ses requêtes et se voit ainsi soulagée de nombreux traitements. De plus, de cette façon, elle peut se consacrer entièrement à ses travaux de formulation de requêtes et de présentation des résultats. Dans notre exemple, la séparation des tâches permet de réduire considérablement le trafic sur le réseau. En effet, seules les dix lignes du résultat de la requête vont transiter par le réseau. Par contre, si

le traitement des données n'était possible que du côté client, il faudrait transférer les 100000 lignes de la table.

Le modèle Client-Serveur de données est un des types de Client-Serveur les plus répandus. Actuellement, pratiquement tous les éditeurs de SGBD relationnels commercialisent un IPC associé à leur système. Ils permettent ainsi l'accès à leur système à partir de n'importe quelle station de travail ou PC connecté sur le réseau.

Le Client-Serveur de procédures

Avec le type de Client-Serveur de procédures, la répartition des tâches entre client et serveur est pratiquement égale. En effet, comme nous le montre la Figure 8, le client prend en charge les deux modules concernant la présentation, ainsi que le module de logique fonctionnelle, tandis que le serveur supporte les trois modules restants.

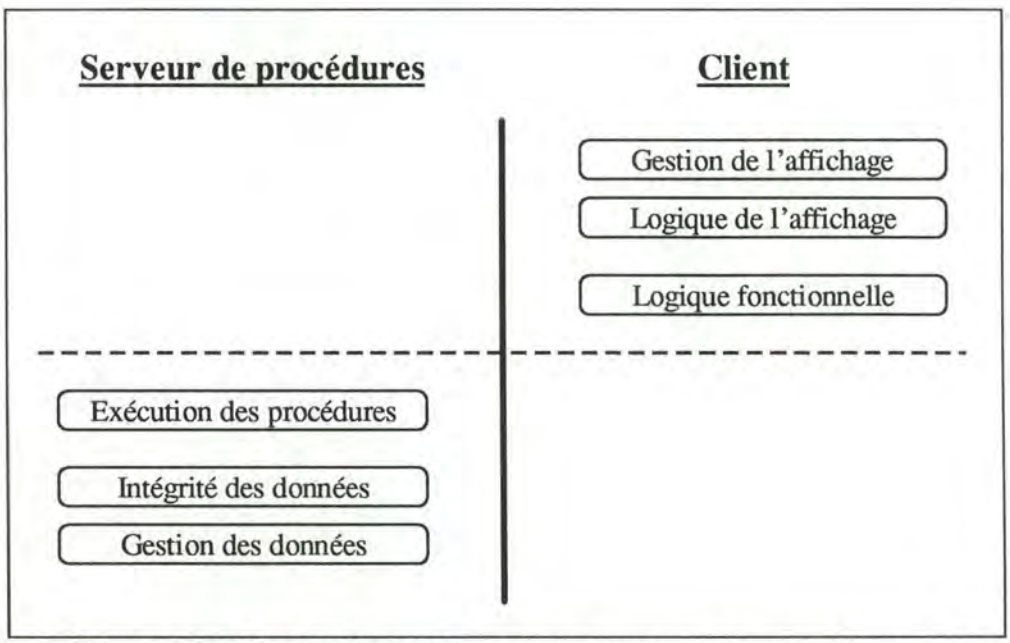

Figure 8 - Répartition des traitements entre un serveur de procédures et un client

Mais cette découpe n'est qu'une représentation de principe du Client-Serveur de procédures. Les trois modules que supporte le serveur ne sont pas indissociables. Seul le module d'exécution des procédures est obligatoirement supporté par le serveur de procédures. Les deux autres, c'est-à-dire ceux concernant la gestion et l'intégrité des données, peuvent se trouver sur un serveur de données séparé. On aura ainsi une situation dans laquelle une application cliente fait appel à un serveur de procédures pour exécuter une partie de ses traitements, et à un serveur de données pour s'occuper de la gestion et de l'intégrité des données. On obtient une répartition des traitements entre client et serveurs symbolisé par la Figure 9.

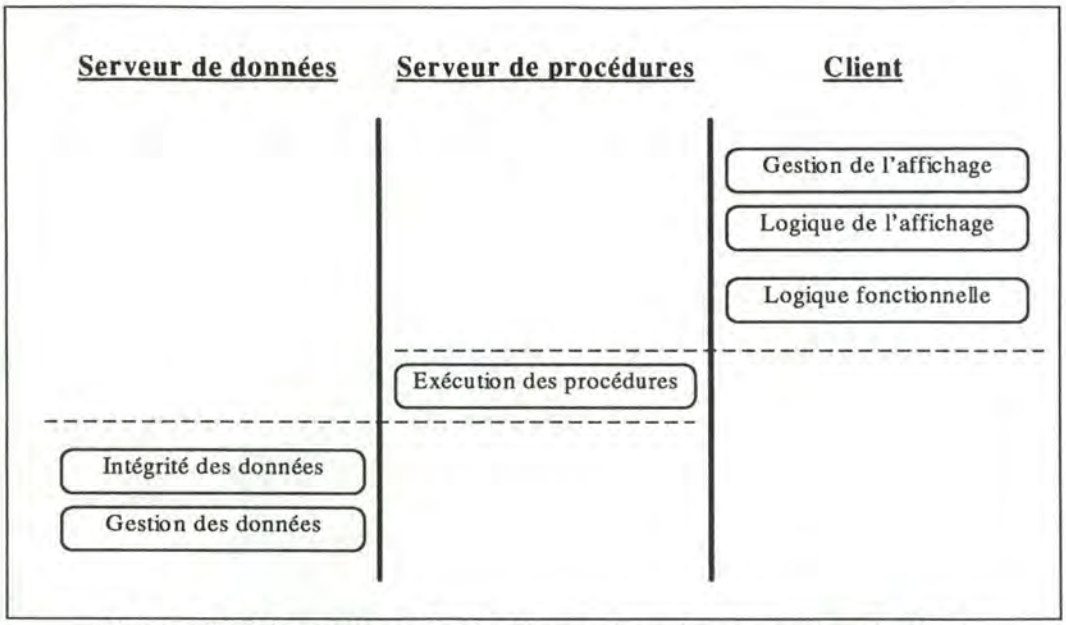

Figure 9 - Répartition des traitements entre serveurs et client

On peut implémenter le Client-Serveur de procédures en utilisant un mécanisme de type Remote Procedure Call (RPC), c'est-à-dire d'appel d'une procédure à distance. Le programme client fait appel à des procédures qui résident et qui seront exécutées du côté serveur. Pour que cela soit possible, l'application doit passer au serveur, lors de son message d'appel, toute une série d'informations permettant à celui-ci d'identifier la procédure à exécuter, de vérifier si le niveau d'autorisation du client demandant l'exécution de la procédure est suffisamment important, et de disposer des données nécessaires au bon fonctionnement de la procédure. Le serveur, quant à lui, va exécuter la procédure et envoyer les résultats au client concerné.

L'utilisation d'un SGBD disposant de procédures stockées rentre aussi dans le contexte du Client-Serveur de procédures. En effet, les applications clientes n'envoient plus de simples requêtes d'interrogation au serveur, mais font appel à des procédures à différents niveaux de complexité stockées sur le serveur.

Dans le cas des RPC, la communication entre Client-Serveur est dite sans connexion. Dès que l'application a fait appel à une procédure du côté serveur, elle est interrompue jusqu'à ce que le serveur lui réponde. On se trouve donc dans le cas d'un échange synchrone. Par contre, avec les procédures stockées, on se trouve en mode connecté, c'est-à-dire dans le cas d'un dialogue avec session. Le client et le serveur s'échangent, en plus des habituelles requêtes et résultats, des points de synchronisation qui vont permettre une communication asynchrone.

Mais quelle que soit la méthode utilisée, on obtient de toute façon une meilleure répartition des tâches entre client et serveur ainsi qu'une diminution de la quantité d'informations échangées sur le réseau.

Si on considère la méthode des procédures stockées, on constate également un gain au niveau de la consommation des ressources. Avec le Client-Serveur de procédures, on fait appel à des procédures déjà précompilées sur le serveur, cela permet en outre d'avoir une grande rapidité d'exécution, et de ne pas avoir à réinterpréter les requêtes comme c'est le cas avec un serveur de données.

Toutefois, le Client-Serveur de procédures ne possède pas que des avantages. Il ne convient pas, par exemple, pour des applications permettant de formuler n'importe quel type d'interrogation sur une base de données. Avec ce type de Client-Serveur un développement préalable des procédures du côté serveur s'avère indispensable.

1.4. Résumé

Pour fixer les idées, et pour nous permettre de percevoir le caractère combinatoire des différents types de Client-Serveur, nous allons conclure par un exemple d'application exploitant au maximum les possibilités de ce modèle.

Soit une application cliente structurée de telle façon que ses modules de gestion de l'affichage, d'exécution des procédures et de gestion et d'intégrité des données soient supportés respectivement par des serveurs d'affichage, de procédures et de données. Le coeur de l'application constitué des modules logique de l'affichage et logique fonctionnelle s'exécute sur la machine cliente. Le module de gestion de l'affichage est pris en charge par un serveur X de présentation avec lequel l'application communique à l'aide du protocole X11. L'application utilise ensuite un mécanisme de type RPC pour exécuter des procédures à distance. Ces procédures sont abritées sur un serveur dédié, qui est donc un serveur de procédures. L'exécution de ces procédures peut entraîner l'envoi de requêtes SQL vers un serveur de données. La demande d'exécution d'une procédure par l'application cliente se voit ainsi prolonger par la formulation de requêtes SQL destinées à un serveur de données.

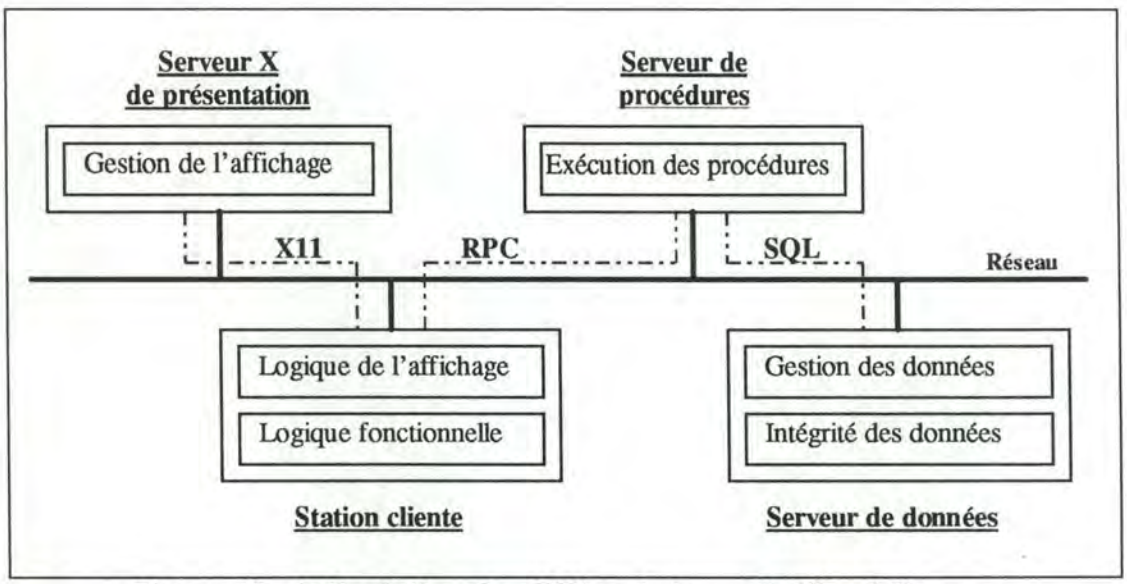

Figure 10 - Articulation des différents types de Client-Serveur

Il est maintenant évident que les différents types de Client-Serveur ne sont pas exclusifs, et peuvent donc se combiner et se compléter entre eux.

Toutefois, la façon d'architecturer une application en utilisant les différents types de Client-Serveur ne se limite pas à celle présentée dans la Figure 10. En effet, on aurait pu avoir une découpe de l'application telle que certaines des procédures appelées soient générales, et peuvent donc se trouver sur un serveur de procédures, et d'autres plus spécifiques à l'application,

s'exécutant directement sur la machine cliente. Ces procédures non génériques envoyant également des requêtes SQL à un serveur de données sans passer par le serveur de procédures.

CHAPITRE 2. LE SYSTÈME X-WINDOW: UNE APPLICATION DU MODÈLE CLIENT-SERVEUR DE PRÉSENTATION

Le chapitre 1 nous a permis de présenter de façon générale le modèle Client-Serveur et d'identifier ses différentes mises en oeuvre, à savoir les Clients-Serveurs de données, de procédures et de présentation. Nous l'avons vu, avec ce dernier type de Client-Serveur, les applications clientes doivent être capables de communiquer avec un serveur d'affichage suivant un protocole pré-défini, les échanges conformes à ce protocole devant pouvoir circuler sur un réseau. A l'heure actuelle seul le système X-Window répond à cette définition. Le chapitre 2 va nous permettre de présenter en détail ce système particulier.

2.1. Qu'est-ce que le système X-Window ?

Le système X-Window est un système multi-fenêtré multi-plateformes. Il peut s'exécuter aussi bien sur des stations de travail, des mini-ordinateurs, des mainframes, ou des super-ordinateurs. X fournit le moyen à une organisation de disposer, sur l'ensemble des machines connectées sur le réseau, d'une interface graphique, et cela, indépendamment du type des machines et de leur système d'exploitation (cf. Figure 11)[4].

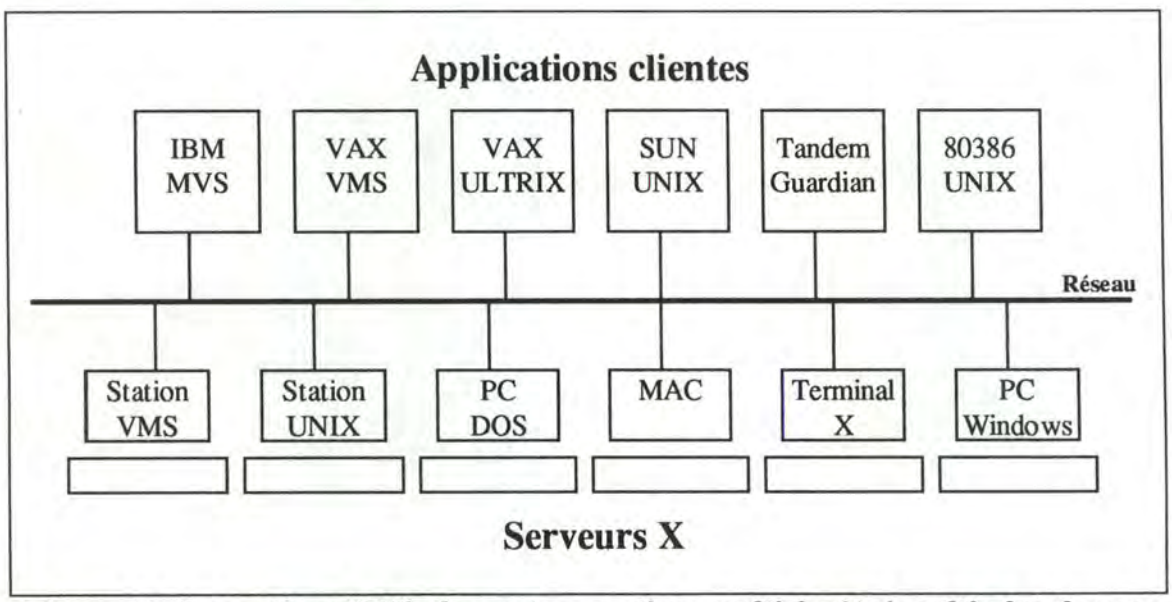

Figure 11 - Le système X-Window est un système multi-fenêtré multi-plateformes

[4] MANSFIELD N., *The joy of X*, Addison-Wesley, 1992, p. 27.

Avant d'aborder la présentation des caractéristiques générales de X, revenons quelque peu en arrière, et intéressons-nous à l'historique de ce système.

Le système X-Window a été développé dans les laboratoires du service informatique du Massachusetts Institute of Technology (MIT), avec le soutien de Digital Equipment Corporation (DEC) dans le cadre du projet Athena. Le nom de X, ainsi que certaines caractéristiques de base, provient d'un système multi-fenêtré plus ancien développé par Brain Reid et Paul Asente de l'université de Stanford, qui s'appelait W.

Les premières versions ont été utilisées uniquement au sein du MIT et de DEC, mais à partir de la version 10, de nombreux constructeurs ont exprimé le souhait de voir commercialiser le système. C'est en 1986, avec la version 10 révisée pour la quatrième fois (mieux connue sous le nom X10R4), que X est devenu disponible au public.

En septembre 1987, la version 11.1 est apparue sur le marché. Elle offrait de meilleures performances que la version précédente, était extensible, supportait la gestion de plusieurs écrans ainsi que différents styles de gestionnaire de fenêtres. Mais une des caractéristiques les plus importantes était qu'elle garantissait la compatibilité entre elle-même et ses futures révisions, ce qui n'était pas le cas jusqu'alors.

Avec la version X 11.2 en mars 1988, le X Consortium était créé. Ce Consortium, qui avait pour but de définir et de maintenir les standards du système, regroupait la plupart des grands constructeurs informatiques. Les versions 11.3 et 11.4 furent respectivement disponibles en février 1989 et en janvier 1990. A l'heure actuelle, c'est la version 11.6 qui est utilisée.

Au fil du temps, la distribution du MIT n'a cessé de s'étendre, pour offrir aujourd'hui une large gamme de services.

Actuellement, les éléments qui permettent à X de se différencier des autres systèmes sont les suivants :

- Contrairement aux autres systèmes multi-fenêtrés, X ne définit pas un style d'interfaçage particulier. Par contre, il fournit des mécanismes qui autorisent différents types d'interfaces. Le choix est donc laissé aux programmeurs d'applications d'opter pour tel ou tel style d'interfaçage.

- Lorsqu'un programmeur développe une application sous X, il a l'assurance que son programme pourra être utilisé dans tous les environnements suivant le protocole X. En effet, une des caractéristiques du système X-Window est son architecture complètement indépendante des périphériques. Un programme développé sous X a la possibilité d'afficher des fenêtres contenant du

texte ou des graphiques sur n'importe quelle machine respectant le protocole X sans avoir à recompiler ou à relier l'application.

- De plus, c'est un système multi-fenêtré orienté réseau. Le système X-Window permet d'avoir des applications s'exécutant sur une machine, et des utilisateurs, utilisant ces applications, situés sur d'autres machines connectées sur le réseau.

En résumé, on peut dire que toutes les applications développées sous X sont disponibles pour tous les utilisateurs, et cela indépendamment du type de plate-forme sur laquelle ils se trouvent. X est un véritable système ouvert en action.

2.2. L'architecture Client-Serveur de X

Le système X-Window possède une architecture de type Client-Serveur, il respecte donc l'ensemble des principes compris dans le modèle du même nom. A savoir, la présence, d'une part, des deux composants principaux, le client et le serveur et, d'autre part, d'un canal de communication reliant ces deux éléments et permettant l'établissement d'un dialogue entre eux.

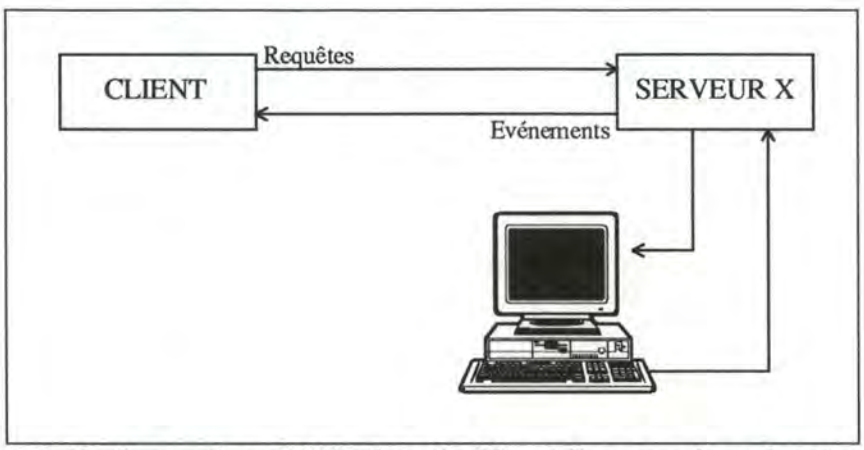

Figure 12 - Le système X-Window, le Client-Serveur de présentation

Remarquons que le type de Client-Serveur de X, c'est-à-dire celui de présentation, ne permet pas toutes les configurations d'utilisation qui sont possibles avec les types de Client-Serveur de données et de procédures.

Avec X, seules deux configurations sont admises. Le serveur doit obligatoirement s'exécuter sur la machine à laquelle est connectée le périphérique partagé. L'application cliente est libre, quant à elle, de s'exécuter sur la même machine ou sur une autre à distance. Ce dernier cas de figure est caractéristique des terminaux X.

Les Clients-Serveurs de données et de procédures permettent un plus grand nombre de configurations d'utilisation puisqu'ils autorisent le client et le serveur de s'exécuter indifféremment à distance ou localement. On peut donc avoir les situations suivantes: soit le client et le serveur s'exécutent tous deux à distance ou localement, soit l'un des deux réside sur la machine locale tandis que l'autre s'exécute autre part sur le réseau. Quatre configurations sont donc possibles avec ces types de Clients-Serveurs.

Cette précision étant faite, intéressons-nous maintenant aux rôles et aux caractéristiques du serveur et des clients.

Le serveur est un programme qui contrôle l'ensemble des périphériques d'entrée et de sortie d'un poste de travail, c'est-à-dire le ou les écrans, le clavier et la souris. Le client est quant à lui une application faisant appel aux services offerts par le serveur.

Le serveur doit toujours être à l'écoute des demandes de services formulées par les clients. Ces demandes sont formulées sous la forme de requêtes correspondant chacune à un service offert par le serveur. On aura par exemple une requête pour la création d'une fenêtre, une pour son déplacement, une autre pour changer sa taille, ou encore pour y dessiner du texte et des graphiques. D'un autre côté, le serveur veille à faire savoir aux clients les différentes entrées en provenance du clavier et de la souris. Pour ce faire, il recourt à des événements. Chaque événement correspond à une action particulière de l'utilisateur (taper un caractère, déplacer la souris, appuyer ou relâcher un bouton de la souris, etc.) ou à un changement de l'état des fenêtres (une fenêtre auparavant cachée se trouve à nouveau exposée par exemple).

Avant de pouvoir communiquer avec un serveur, un client doit demander une connexion avec ce dernier. Une fois cette connexion établie, l'application cliente peut accéder à l'ensemble des écrans contrôlés par ce serveur, et l'échange de requêtes et d'événements peut donc commencer. Cette communication est contrôlée par le protocole de X, qui spécifie entre autres le format des paquets échangés sur le réseau. Si une application respecte le protocole X, alors elle peut se connecter à n'importe quel serveur disponible sur le réseau. On peut même avoir une situation dans laquelle un client est connecté à plusieurs serveurs, et où plusieurs applications clientes sont connectées à un serveur unique. Toutefois, le système X-Window fournit un certain nombre de mécanismes de sécurité qui donnent la possibilité à un serveur de refuser une connexion émanant d'une application cliente s'exécutant sur une autre machine.

Ce qu'il ne faut pas perdre de vue, c'est que le serveur et le client sont tous deux des programmes distincts qui peuvent donc être exécutés soit sur la même machine, soit sur des machines séparées. Dans le premier cas, la communication entre client et serveur est toujours contrôlée par le protocole X, mais l'accès au réseau n'est plus nécessaire. En effet, dans un souci d'optimisation des performances, on utilisera des techniques de communication inter-processus telles que les mécanismes de mémoires partagées. Dans le second cas, on aura le serveur qui s'exécute sur la machine de l'utilisateur, et le client sur une autre machine connectée sur le réseau. L'exécution à distance du programme client est totalement transparente pour l'utilisateur. Tout se passe, en effet, comme si l'application cliente s'exécutait sur sa propre machine.

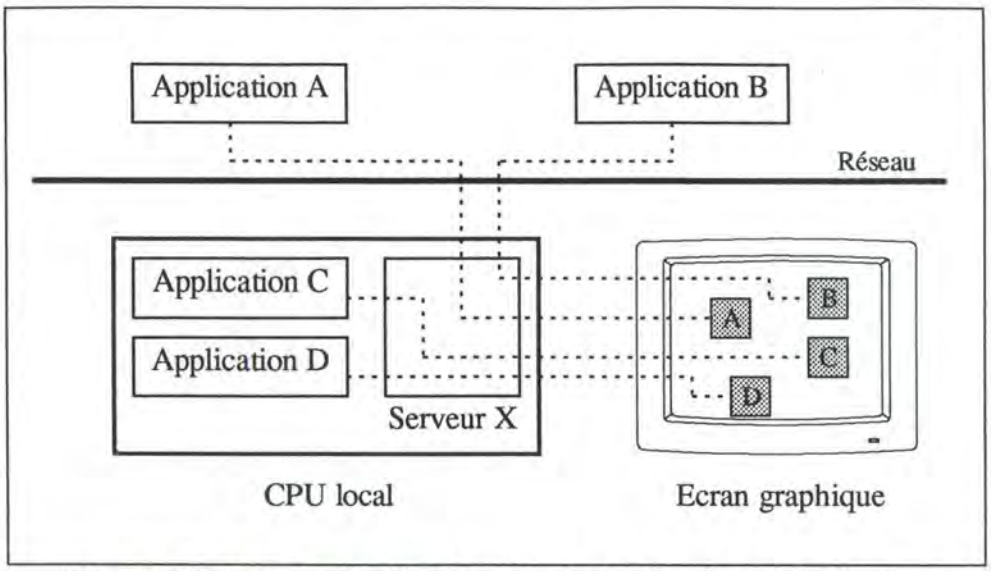

Figure 13 - Les applications peuvent être locales ou éloignées

La Figure 13 nous donne une illustration d'applications clientes s'exécutant sur une machine locale et sur des machines éloignées. On a un serveur qui s'exécute sur une machine et qui contrôle donc son écran. Ensuite, on trouve les applications C et D locales à la machine abritant le serveur. On a enfin les applications A et B qui s'exécutent sur des machines quelque part sur le réseau. Toutes ces applications sont connectées au même serveur, et lui envoient donc leurs requêtes soit via le réseau, soit par des mécanismes internes à la machine. Chaque application accède ainsi au même écran. Pour l'utilisateur manipulant ces applications, rien ne lui permet de faire la distinction entre les applications s'exécutant localement, et celles s'exécutant à distance.

On vient de voir que X laisse le choix à un utilisateur d'exécuter une application à distance ou sur sa propre machine. Cette propriété de X est seulement utile aux utilisateurs disposant d'une station de travail, mais est indispensable à ceux travaillant avec un terminal X. En effet, une station de travail permet le multitâche, et peut donc abriter, en plus du serveur X, plusieurs applications actives simultanément. Par contre, lorsque l'on dispose d'une machine munie d'une souris, d'un clavier, d'un écran graphique, d'une interface réseau et d'un serveur X en mémoire morte (ROM), c'est-à-dire d'un terminal X, seul le serveur peut être exécuté localement. Les applications clientes s'affichant sur un terminal X se voient donc dans l'obligation de s'exécuter sur une autre machine.

Les avantages que procurent l'utilisation d'un terminal X sont nombreux. Ils sont d'abord financiers. En effet, un terminal X, de par sa simplicité logicielle et matérielle, est nettement moins cher qu'une station de travail. Pourtant ces deux machines vont disposer de la même interface, et pourront exécuter les mêmes applications clientes. Ensuite, un terminal X n'est pourvu que d'un système d'exploitation minimum, il pourra donc facilement s'adapter sur un réseau. Enfin, le redémarrage d'un terminal X après des ennuis techniques ayant

entraîné l'arrêt du système est simple, vu qu'aucun système de fichiers ne doit être restauré, un terminal X ne disposant pas d'unité de stockage.

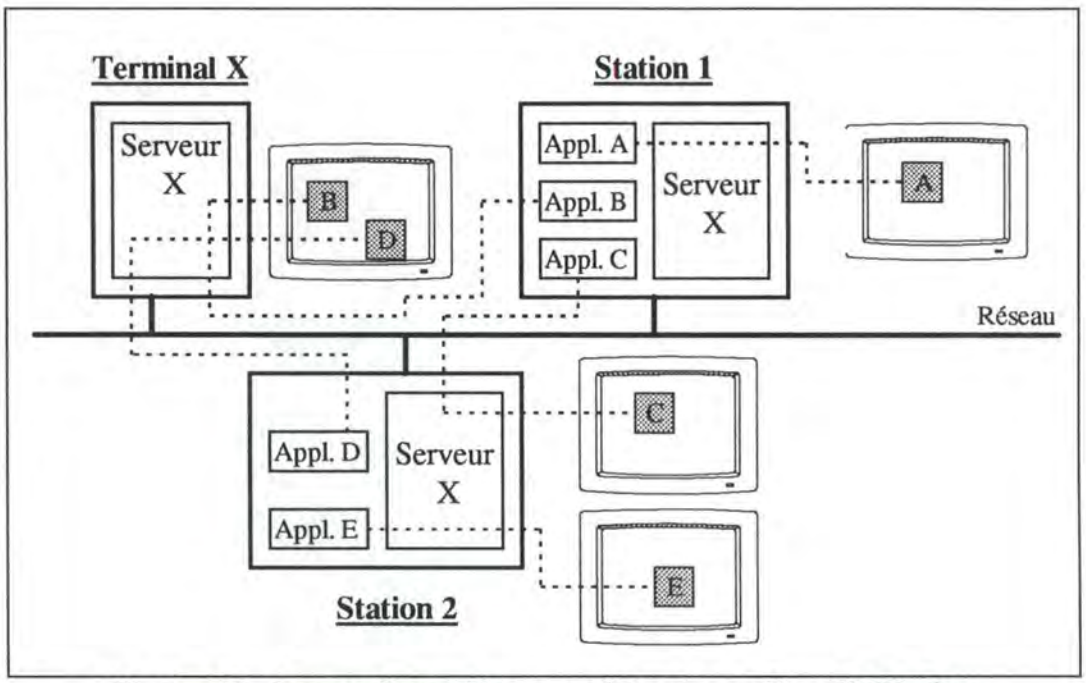

Figure 14 - Les configurations possibles du système X-Window

La Figure 14 illustre bien les architectures et les utilisations que le système X-Window rend possibles. On retrouve connectés sur un même réseau, des stations de travail et un terminal X. Chacun d'eux abrite un serveur X qui contrôle un ou plusieurs écrans. Les applications s'affichant sur le terminal X sont bien entendu exécutées à distance (applications B et D), tandis que celles des stations de travail sont soit locales (applications A et E), soit éloignées (application C).

2.3. Comparaison avec d'autres systèmes

Pour se convaincre des qualités et des avantages que procure le système X-Window, il suffit de le comparer aux autres systèmes disponibles sur le marché. Ces derniers possèdent chacun des qualités individuelles, mais seul X regroupe l'ensemble de ces caractéristiques pour former un système souple et puissant répondant aux besoins actuels des organisations.

Telnet

TELNET est un protocole de niveau application qui permet à deux processus situés sur des machines différentes et munies de systèmes d'exploitation différents de communiquer à l'aide d'un langage commun sur une voie de communication TCP.

Le programme telnet permet à un utilisateur de se connecter, à partir de sa propre machine, sur n'importe quelle autre machine présente sur le réseau. Ensuite, une fois que cette connexion est établie, l'utilisateur peut exécuter des processus sur cette machine distante, en ayant l'affichage des résultats sur sa propre machine. Toutefois, il est important de souligner que telnet est limité aux applications orientées caractères.

Tout comme X, telnet offre un service d'exécution de programmes à distance. Ce service est cependant limité aux applications orientées caractères, et n'englobe donc pas les programmes disposant d'une interface utilisateur graphique.

Microsoft Windows 3.x pour DOS

Microsoft Windows est un système multi-fenêtré pour PC muni d'une interface utilisateur graphique directement construite au sein du système. Pour cette raison, l'interface utilisateur de Windows ne peut être facilement modifiée. Cette caractéristique le met en désaccord avec le système X-Window, qui donne la possibilité aux utilisateurs de choisir l'interface qui leur convient le mieux. De plus, avec Microsoft Windows, les utilisateurs ne pourront exécuter que des applications locales et spécialement écrites pour un PC DOS. Cependant, des point communs peuvent être trouvés entre les deux systèmes, notamment le fait qu'ils sont tous deux des programmes optionnels qui ne font pas partie du système d'exploitation.

Apple Macintosh

Le Macintosh se distingue du système Microsoft Windows de par sa position vis-à-vis du système d'exploitation. En effet, ce système multi-fenêtré fait partie intégrante et est construit en plein coeur du système d'exploitation. L'utilisation d'un Macintosh passe donc inévitablement par la manipulation de son système multi-fenêtré.

Le Macintosh dispose d'une interface utilisateur graphique difficilement modifiable ainsi que la possibilité d'avoir plusieurs applications actives simultanément.

Systèmes développés par Sun

SunView, qui est le système multi-fenêtré de SUN (Stanford University Network), dispose de caractéristiques identiques à celles du Macintosh. Cependant, SUN, qui est un grand constructeur de stations de travail, a développé un système disposant des principaux avantages de X. Ce système, qui s'appelle NeWS (Network extensible Window System), est basé sur le modèle Client-Serveur. Mais il est la propriété de SUN, et n'est pratiquement utilisé que par lui. Par contre, avec son système OpenWindows, qui est une combinaison de NeWS et de X, SUN compte bien se faire une place privilégiée dans le monde du Client-Serveur.

A l'heure actuelle, le système X-Window est adopté par les plus grands constructeurs informatiques tels que SUN, Digital, IBM et Hewlett-Packard. Grâce à une combinaison unique des qualités présentes individuellement dans les autres systèmes, X s'impose de plus en plus comme un standard dans le monde des systèmes multi-fenêtrés. Un exemple, Digital a choisi X pour être son système multi-fenêtré : DECwindows.

2.4. Les ressources de X

Un serveur X gère les *ressources* des clients. Ces ressources sont: les fenêtres, les bitmaps, les polices de caractères, les couleurs, etc. Elles se présentent sous forme de structures, allouées par le serveur, auxquelles les applications clientes font référence par un numéro d'identification attribué de manière unique par le serveur.

La fenêtre

Une fenêtre est une région rectangulaire de l'écran dans laquelle il est possible de dessiner. Les fenêtres de X ont ceci de particulier: contrairement à d'autres systèmes de fenêtrage, elles n'ont ni barre de titre, ni ascenseur, ni aucune autre décoration. Elles n'ont qu'un bord. Il incombe à l'application cliente de créer autant de fenêtres que nécessaire pour pouvoir simuler une barre de titre ou un ascenseur. Par exemple, un menu déroulant est constitué d'une fenêtre délimitant les contours du menu et d'une série de fenêtres représentant les choix de ce menu.

Une fenêtre, appelée *fenêtre fille*, peut être imbriquée dans une autre fenêtre, appelée *fenêtre mère*, et une fenêtre peut être recouverte par d'autres. Toutes les fenêtres sont organisées hiérarchiquement entre elles, sous forme de structure en arbre (cf. Figure 15). Le sommet de cette hiérarchie est représenté par l'écran lui-même, appelé *fenêtre racine*. Une fenêtre est donc spécifique à un écran.

La hiérarchie des fenêtres dicte les règles de visibilité d'une fenêtre. Ainsi, une fenêtre fille peut être plus grande que la fenêtre mère, mais les parties situées en dehors des limites de la fenêtre mère seront invisibles, coupées (*clipped*).

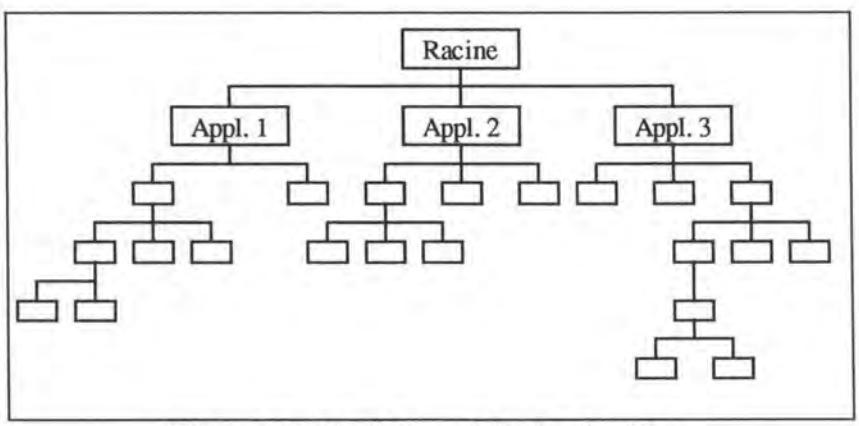

Figure 15 - La hiérarchie des fenêtres

La pixmap

Une pixmap est un tableau à deux dimensions de pixels, chaque pixel consistant en un certain nombre de bits représentant la *profondeur* du pixel. En général, tous les pixels d'un écran ont la même profondeur. Cette profondeur détermine la couleur d'un pixel. Par convention, une pixmap de profondeur une est appelée *bitmap*.

En fait, une pixmap est identique à une fenêtre, sauf qu'elle n'est pas visible à l'écran. Ainsi, toutes les commandes de tracé sont applicables aussi bien à une fenêtre qu'à une pixmap (excepté deux commandes spécifiques à chacun des deux types, **XClearWindow** et **XClearArea**). Des régions d'une pixmap peuvent être copiées dans une fenêtre, et vice-versa, à condition que ces deux ressources aient la même profondeur et appartiennent au même écran.

Une pixmap est souvent utilisée pour spécifier le motif du fond d'une fenêtre. Elle est aussi utilisée lorsqu'une fenêtre doit être redessinée à l'écran. La fenêtre peut avoir été dessinée par l'appel d'une centaine de primitives graphiques. Redessiner cette fenêtre après qu'elle ait été cachée par une autre peut alors s'avérer coûteux en temps de calcul. La pratique couramment utilisée est de dessiner la même chose dans une fenêtre ET dans une pixmap. Lorsque la fenêtre doit être redessinée, il suffit alors de copier la pixmap dans la fenêtre. L'inconvénient est que ces pixmaps utilisent de la mémoire et que l'ouverture de plusieurs applications peut rapidement mener à la saturation d'un serveur disposant de peu de mémoire.

Les types de ressources *fenêtre* et *pixmap* sont aussi connus sous l'appellation *drawable*.

La police de caractère

Une police de caractères est un ensemble de bitmaps, gérées du côté serveur, chacune déterminant l'apparence d'un caractère à l'écran. Chaque police de caractère possède un nom normalisé appelé *X Logical Font Description* (*XLFD* - Voir la spécification Figure 16). Ce nom contient des informations telles que la famille, la hauteur, la largeur moyenne, les résolutions verticale et horizontale, l'inclinaison, etc. Ainsi, le serveur ne doit pas nécessairement charger la police pour transmettre aux clients des informations sur cette police. Le serveur accepte des noms génériques (par l'intermédiaire du joker '*') pour faciliter la recherche d'une police de caractère. Par exemple, une application peut vouloir charger n'importe quelle police, pour autant qu'elle soit italique et qu'elle ait une hauteur de 12 points.

Les polices de caractères X sont stockées sur disque dans le format *Portable Compiled Format* (PCF) portable sur toutes les plateformes X11R5. Les polices peuvent aussi être distribuées dans le format *Bitmap Distribution Format*

(BDF), simple fichier ASCII, qui doit alors être converti en format PCF, au moyen d'un compilateur livré avec le serveur.

Les polices de caractères peuvent être codées sur un ou deux bytes. Une police codée sur 8 bits acceptera au plus 256 caractères, ce qui est suffisant pour les polices des pays européens. Mais, certaines polices, comme les polices orientales, ont besoin de plus de 256 caractères. Elles utilisent alors un codage sur 16 bits, autorisant ainsi 65536 caractères. L'accès à un caractère se fait par la position ou l'indice du caractère dans sa police. Ainsi, la position de D dans le vecteur d'encodage ASCII est 68. Mais, D, dans un autre vecteur d'encodage, peut avoir une position différente. La dernière partie du nom XLFD spécifie ce vecteur d'encodage. Dans notre exemple, la police utilise le vecteur d'encodage ISO 8859-1 qui signifie *Alphabet Latin ISO numéro 1.* Il existe ainsi d'autres ISO 8859 pour les alphabets cyrillique, hébreu, grec, etc.

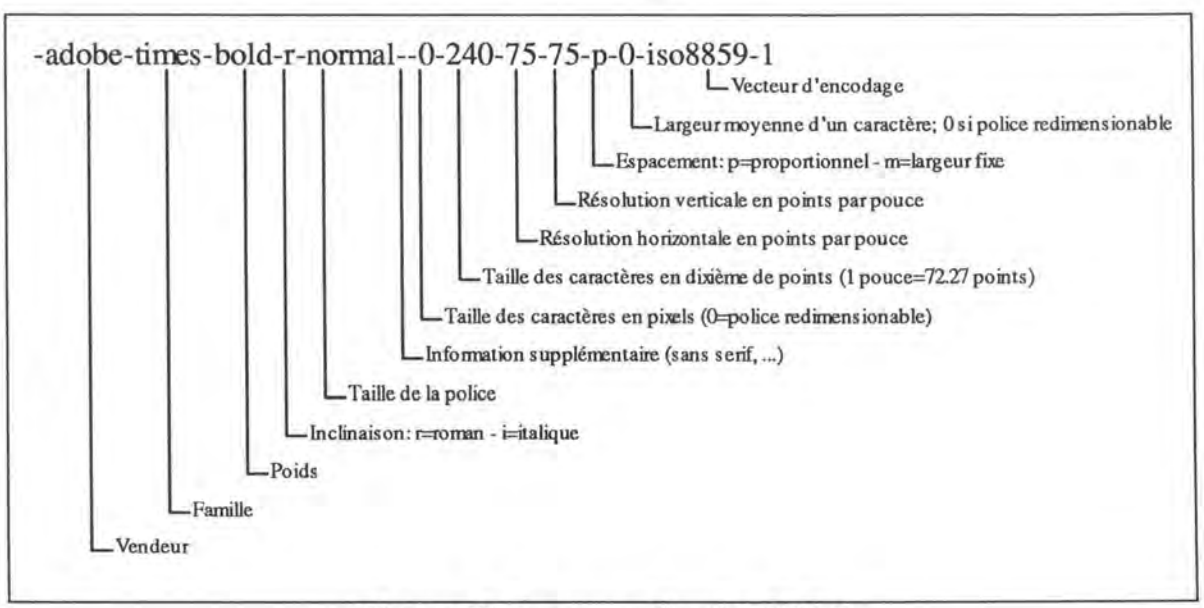

Figure 16 - La normalisation XLFD

Avant la version 5 de X, il fallait un fichier de description différent pour une même police mais pour des hauteurs de caractères différentes puisqu'elle était représentée par des bitmaps. Il existe bien sûr un moyen de donner n'importe quelle taille à une police bitmap, mais le résultat présente un effet d'escalier. La solution de ce problème réside dans la façon de stocker les caractères. En effet, un caractère peut être conservé sous forme d'un ensemble de courbes (ou de vecteurs) dont la taille peut être facilement modifiée et qui peut être ensuite transformé en une image bitmap. Ce sont des *polices vectorielles*. Le système Bitstream Speedo intégré à X11R5 est une implémentation de ces polices. Les polices Type 1 en sont une autre.

Le stockage de polices de caractères sur chaque serveur X est une source de problèmes d'administration. Tout d'abord, la place requise par les polices de caractères pour un serveur, sur disque ou en mémoire morte (dans le cas d'un

terminal X), multipliée par autant de serveurs présents sur le réseau, peut être très importante. De plus, la mise à jour des polices s'avère fastidieuse, puisqu'elle doit se faire sur tous les serveurs. Ensuite, certaines polices vectorielles sont très gourmandes en CPU et peuvent ralentir fortement une petite machine. C'est pour remédier à ces problèmes que le concept de *serveur de polices de caractères* a été développé. Un serveur X, lorsqu'il a besoin d'une police de caractère, envoie une requête à un serveur de polices qui la lui renvoie dans le format bitmap demandé. Les serveurs X peuvent se connecter à plusieurs serveurs de polices et plusieurs serveurs de polices peuvent être chaînés. Ainsi, si un serveur X demande une police à un serveur de polices qui ne la possède pas, celui-ci peut s'adresser à un autre serveur de polices. On note ainsi qu'avec une telle architecture, il n'y a plus de redondance inutile: une police de caractères se trouve à un seul endroit; l'ajout d'une nouvelle police de caractère se fait très rapidement: il suffit de l'ajouter au serveur de polices de caractères; le changement de format de stockage d'une police de caractères peut se faire sans problème, puisqu'il suffit de modifier le serveur de polices uniquement; et le serveur de polices de caractères peut être installé sur une machine puissante qui pourra traiter rapidement les polices vectorielles.

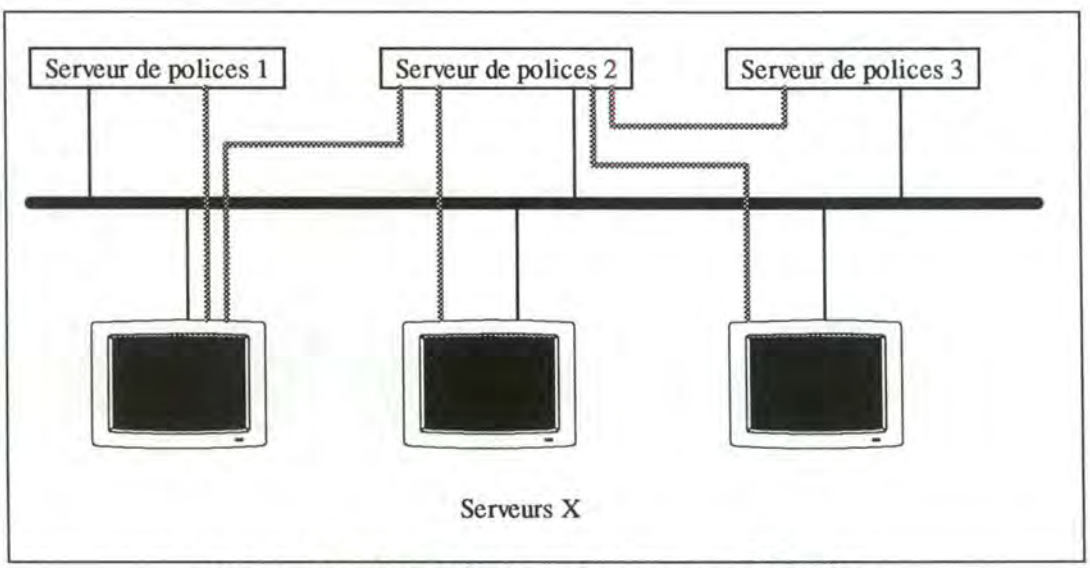

Figure 17 - L'architecture des serveurs de polices

La carte des couleurs (colormap)

Sur les systèmes couleurs, la profondeur de chaque pixel de l'écran est souvent inférieure à la profondeur réellement disponible sur cet écran. Par exemple, des écrans peuvent afficher 16,7 millions par pixel. Mais, pour diverses raisons, la mémoire de l'écran n'offre qu'un seul octet par pixel, c'est-à-dire une profondeur de 8 correspondant à 256 (= 2^8) couleurs[5].

[5] Une mémoire écran de profondeur 8 est constituée de 8 plans, chacun correspondant à tous les pixels de même indice.

Une colormap est un tableau dont les éléments constituent une palette de 256 couleurs parmi la totalité des 16,7 millions de couleurs disponibles. Ainsi, si un pixel d'une fenêtre spécifie *x* comme couleur, en fait, la couleur réellement utilisée sera celle spécifiée dans l'élément de la colormap indexé par *x*, soit *y* (voir Figure 18). En cours d'utilisation, si *y* est changé en *z* dans la colormap, tous les pixels référençant l'élément d'index *x* seront changés automatiquement. Cette fonctionnalité est particulièrement utile dans les applications de traitement d'images où le résultat d'un changement de contraste, par exemple, doit être visible immédiatement. Notons qu'un élément de la colormap est constitué de trois valeurs de 16 bits chacune, représentant une couleur codée en RGB (Red, Green, Blue ou Rouge, Vert, Bleu).

Chaque application peut avoir sa propre carte des couleurs. Dès qu'une application devient active, le gestionnaire des fenêtres charge la colormap de cette application. Ainsi, si deux applications ont des cartes de couleurs différentes, les couleurs à l'écran de l'application inactive ne seront pas correctes, mais le deviendront dès que l'application sera de nouveau active.

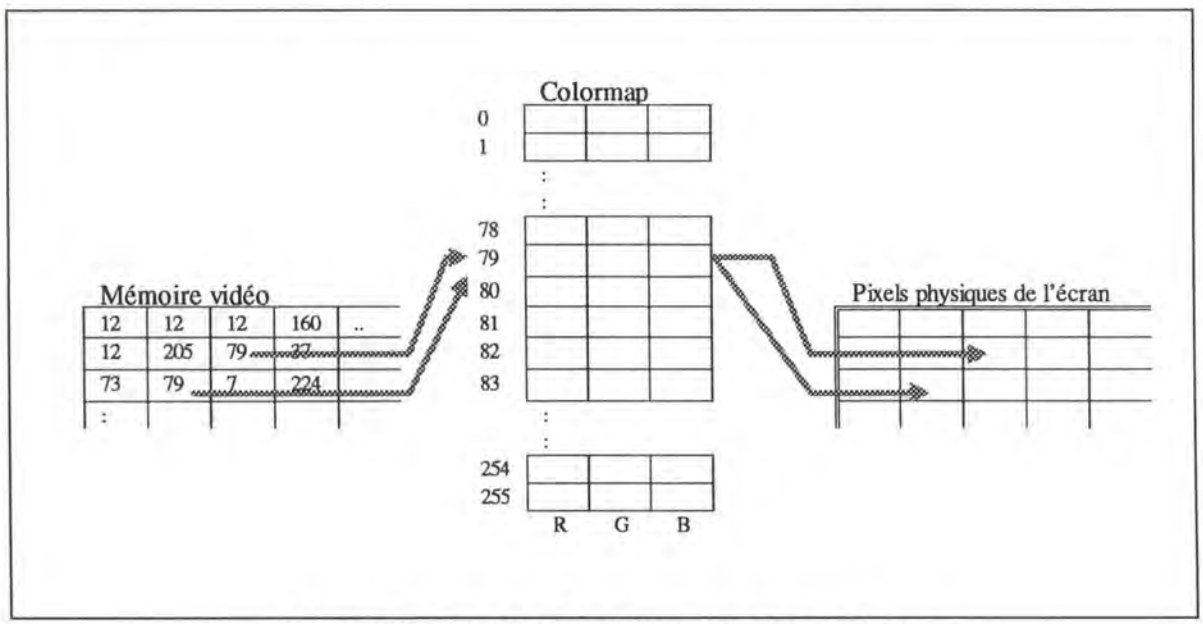

Figure 18 - Le principe de la carte des couleurs

Le curseur

Un curseur est un pointeur à l'écran, dont le mouvement suit celui de la souris. L'apparence de ce pointeur, qui peut changer selon la région au-dessus de laquelle il se trouve, est choisie parmi un ensemble de bitmaps (une police de caractères spéciale). A un curseur est aussi associé un point sensible (*hot spot*), c'est-à-dire un pixel qui précise où se situe exactement le curseur, celui-ci pouvant se situer à cheval sur plusieurs régions de l'écran alors qu'un pixel ne pouvant faire partie que d'une seule région.

Le contexte graphique (GC)

Un contexte graphique détermine la manière dont un objet est dessiné lors de l'appel d'une fonction de tracé. Ainsi, un contexte graphique décide de la couleur, de l'épaisseur, du motif utilisé pour le fond, la façon dont les lignes d'un rectangle se joignent, etc.

Une application, ayant configuré un contexte graphique, peut ainsi l'utiliser pour tracer plusieurs objets de mêmes caractéristiques. L'application peut, bien sûr, créer plusieurs contextes graphiques et les utiliser de manière interchangeable. Notons toutefois qu'un GC est créé en spécifiant une pixmap ou une fenêtre comme référence. Le GC hérite de deux caractéristiques de sa référence: l'écran et la profondeur (voir la carte des couleurs). En conséquence, le GC ne peut être utilisé qu'avec des pixmaps et des fenêtres du même écran et de même profondeur. Ces caractéristiques sont utiles car un GC contient une liste des adresses des fonctions de dessins (cf. Figure 19). Ces fonctions varient selon l'écran et sa profondeur (la mémoire vidéo des écrans ne se gèrent pas tous de la même manière, un écran couleur est différent d'un écran monochrome, etc).

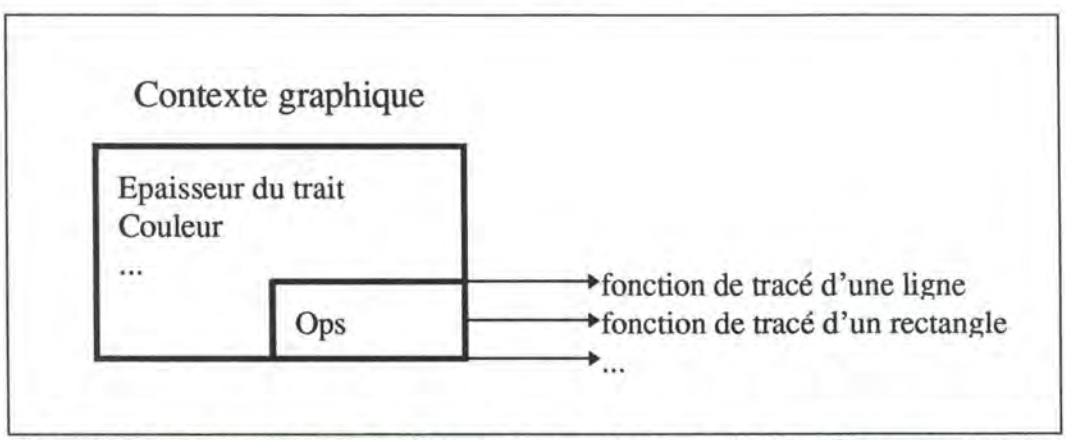

Figure 19 - Le contexte graphique et ses pointeurs de fonctions graphiques

La raison pour laquelle toutes ces ressources sont gérées dans le serveur plutôt que dans le client est une question d'efficacité. En effet, une application cliente peut décider de créer une pixmap, de la garnir d'une série de tracés, puis de la copier dans une fenêtre. Puisque ces deux ressources, pixmap et fenêtre, sont gérées dans le serveur, la copie peut se faire de manière instantanée, sans délai dû à des transferts importants d'informations à travers le réseau. Mais, un des inconvénients de cette fonctionnalité est que la quantité de mémoire vive requise pour faire tourner un serveur est assez élevée. Notons d'ailleurs que cet espace mémoire peut être efficacement géré par l'application cliente en désallouant les ressources devenues obsolètes (par l'appel de fonctions de la la librairie Xlib telles que **XFreeColormap()**, **XFreeCursor()**, etc).

2.5. *Interface de programmation*

En plus du protocole X de dialogue entre une application cliente et un serveur X, une interface de programmation a été définie. Il s'agit de la librairie Xlib, habituellement utilisée à partir du langage C. Cette librairie offre un ensemble de fonctions de bas niveau fournissant un accès et un contrôle de l'affichage et des périphériques d'entrée. Ce sont donc ces fonctions qui s'occupent d'envoyer des requêtes au serveur X. Il existe, par exemple, une fonction pour se connecter à un serveur X, une fonction pour créer une fenêtre, une fonction pour dessiner des lignes dans cette fenêtre, etc.

Ces fonctions étant de bas niveau, une application requiert de nombreuses lignes de code pour présenter à l'utilisateur une interface agréable et intuitive. Il est donc nécessaire d'avoir des boîtes à outils de plus haut niveau. C'est pourquoi il existe des librairies d'objets graphiques sophistiqués (menu, boîte de dialogue, etc.), appelés *widgets*[6]. Le *Xt Intrinsics* est une couche intermédiaire entre la Xlib et les widgets. Xt est une librairie de fonctions permettant la manipulation de ces widgets. Il existe plusieurs ensembles de widgets supportés par Xt. On nommera notamment Athena, Andrew, Motif et Open Look. Du point de vue du programmeur, le passage d'un ensemble de widgets à l'autre se fait sans difficulté puisque les fonctionnalités offertes sont sensiblement équivalentes.

L'utilisation d'une telle boîte à outils a un double avantage. D'abord, il met à la disposition du programmeur un ensemble d'objets pré-programmés qu'il n'a plus qu'à assembler pour confectionner l'interface utilisateur de son application. Ensuite, cela permet d'uniformiser cette interface utilisateur. Ainsi, dans deux applications écrites grâce au même toolkit, l'interface utilisateur sera identique. De plus, cette interface est uniformisée sur toutes les plateformes si le toolkit d'interfaçage est écrit dans un langage portable tel que le C. En effet, un toolkit repose sur une logique d'assemblage d'éléments fournis par la Xlib qui est standardisée et donc potentiellement disponible sur toutes les plateformes. Parmi les toolkits les plus connus, on trouve le toolkit Open Look, le toolkit Motif,... qui offrent chacun une interface utilisateur uniformisée.

L'utilisation d'une boîte à outils de plus haut niveau n'interdit pas l'accès à la librairie de base Xlib. Certains traitements pourraient en effet être plus efficaces par un accès direct à la Xlib. La découpe d'une application se présente alors comme sur la Figure 20: la couche Interface de programmation Xlib qui assure la communication avec le serveur X, la couche Xt Intrinsics qui est une librairie de plus haut niveau et une couche Widgets qui est une librairie d'objets manipulés par la couche Xt.

[6] Contraction de *window* et de *gadget*.

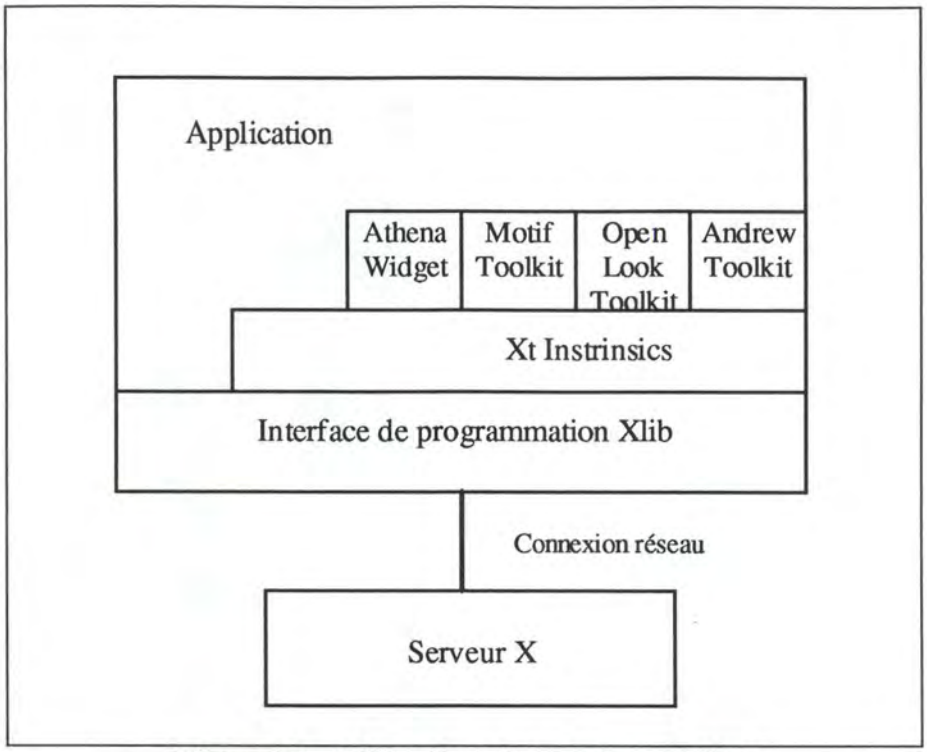

Figure 20 - L'interface de programmation

2.6. *Interface utilisateur*

L'interface utilisateur est en fait composée de deux parties: l'interface application et l'interface de gestion.

L'interface application, utilisée au travers des toolkits précédemment décrits, est l'interface du programme proprement dite et ne peut être altérée sans recompiler le code source avec d'autres libraries. Chaque application peut avoir sa propre interface application puisqu'elle est intégrée à l'application.

L'interface de gestion est concrétisée par un client particulier. Ce client, appelé gestionnaire de fenêtres, contrôle la façon dont sont gérées les fenêtres à l'écran, c'est-à-dire la façon d'agrandir et de diminuer fenêtres, la façon de gérer leur empilement, la façon d'attribuer la main (ou le focus) à une fenêtre, etc. C'est ce programme qui s'occupe de dessiner les contours de la fenêtre, la barre de titre, les icones d'agrandissement et de diminution, etc. Puisqu'il s'agit d'un programme séparé, indépendant du serveur, chaque utilisateur peut choisir son propre gestionnaire de fenêtres. Comme toute application cliente, il peut être exécuté sur une machine distante, mais un seul gestionnaire de fenêtres peut gérer un écran. Il en résulte que toutes les fenêtres d'un écran ont la même interface de gestion. Certains gestionnaires offrent un écran virtuel, c'est-à-dire que l'espace disponible pour les fenêtres est plus grand que l'espace réellement disponible sur l'écran.

L'interface application et l'interface de gestion sont donc totalement indépendantes. Mais, une librairie d'interface application est souvent accompagnée d'un gestionnaire de fenêtres, offrant ainsi tous deux un *look and feel*[7] semblable. C'est le cas notamment de Motif et d'Open Look, les gestionnaires et les librairies les plus couramment utilisés.

[7] Présentation et comportement.

2.7. Requêtes et événements

Lorsqu'une application cliente a besoin d'un service fourni par le serveur X, le client émet une *requête* vers le serveur à travers le réseau. Ces requêtes concernent notamment:

- la création et la suppression des fenêtres;
- la modification des attributs tels que le curseur de la fenêtre, sa bordure ou sa carte des couleurs;
- la configuration d'une fenêtre, c'est-à-dire le changement de sa largeur, sa hauteur, sa position relative vis-à-vis de sa fenêtre parent;
- la demande d'informations à propos d'une fenêtre, de ses attributs et de sa configuration;
- la manipulation et la consultation des propriétés d'une fenêtre (une propriété est une zone de données attribuée à une fenêtre par une application et qui peut être lue par n'importe quelle autre application. Il s'agit donc d'un moyen de communication entre les applications clientes);
- la consultation et le chargement des polices de caractères.

Les requêtes sont basées objet[8], c'est-à-dire que, si une application veut tracer une ligne, elle doit constituer une requête contenant le fait qu'il s'agit du tracé d'une ligne, suivi des coordonnées de début et de fin de la ligne uniquement et non pas tous les points constituant cette ligne. La taille des requêtes est ainsi réduite au minimum, ce qui permet l'utilisation d'une largeur de bande de réseau minimale.

Le serveur et le client agissent tout deux de manière asynchrone. En effet, le serveur, lorsqu'il reçoit une requête, la place dans une file en attendant de pouvoir la traiter. Quand le client reçoit un événement, il le place également dans une file d'attente, jusqu'à ce qu'il ait le temps de le traiter. De même, les requêtes du client sont placées dans une file en attente d'être envoyées. Il est en effet plus intéressant d'envoyer un seul gros paquet sur le réseau plutôt que plusieurs petits paquets. Cet asynchronisme est rendu possible par le fait que la plupart des requêtes n'exigent pas de réponse de la part du serveur[9]. Si le client devait attendre une réponse, le temps d'attente du client équivaudrait au temps d'envoi de la requête + le temps de traitement de la requête par le serveur + le temps de transmission de la réponse. Grâce au fonctionnement asynchrone, le temps d'attente du client se résume au temps d'envoi de la requête.

---

[8] La taxinomie des langages et systèmes orientés objet, présentée dans ZEIPPEN J.-M., *OBLOG - An Objet-Oriented, Formal, Tool-Based Software Engineering Approach*, Institut d'Informatique, FUNDP Namur, March-April 1994, p. 28, étant la suivante: basés objet (les objets sont des unités encapsulées), basés classe (les objets peuvent être instantiés à partir d'une classe), orientés objet (des sous-classes peuvent être dérivées par héritage) et complètement orientés objet (tout est objet, y compris les classes et les valeurs).

[9] Notamment, aucun accusé de réception (ACK) n'est exigé puisque le protocole X fonctionne sur une connexion réseau fiable.

Néanmoins, certaines requêtes requièrent une réponse. Il s'agit des requêtes de demande d'informations. L'utilisation de telles requêtes ralentit le client, puisqu'il doit envoyer la requête immédiatement (en fait, la placer dans la file et envoyer immédiatement le contenu de la file) et attendre que le serveur la traite et lui envoie la réponse.

La Figure 21 montre le format d'une requête d'affichage d'un texte dans une fenêtre. Une requête commence toujours par un code opératoire majeur (*major opcode*) qui identifie le type de requête. Viennent ensuite le nombre d'octets à afficher, la taille de la requête, l'identificateur de la fenêtre dans laquelle le texte doit être affiché, l'identificateur du contexte graphique, les coordonnées (relatives au début de la fenêtre) où doit être affiché le texte et enfin le texte lui-même[10].

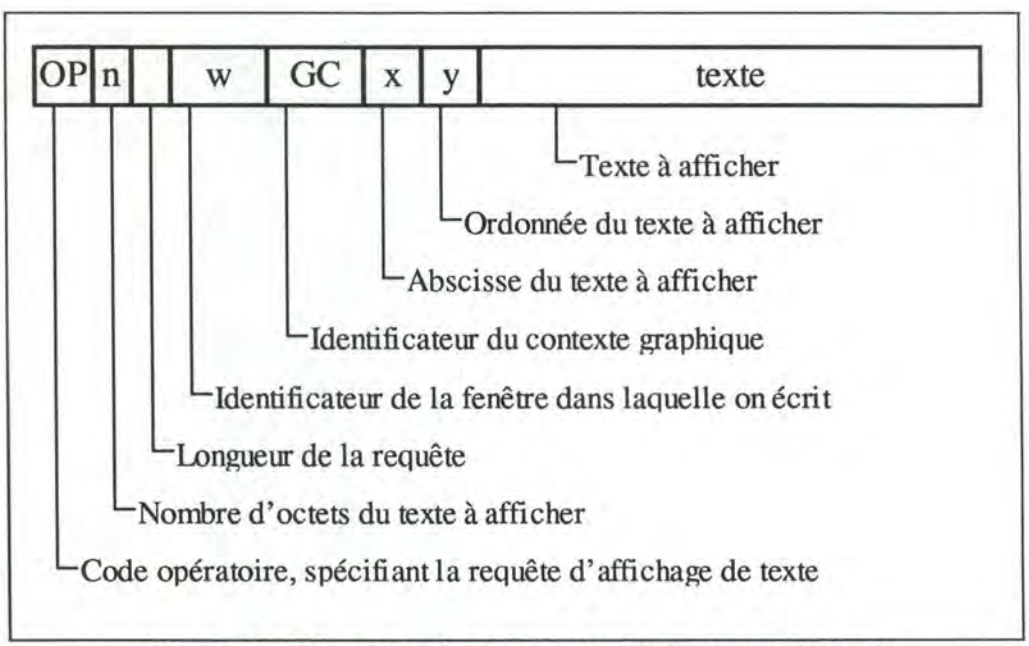

Figure 21 - Exemple d'une requête d'affichage de texte

Comme dans tout système de fenêtrage contrôlé par souris, une application cliente X doit pouvoir répondre à toute une série *d'événements*. Les événements sont envoyés par le serveur pour signaler au client que quelque chose a changé. Enfoncer une touche, cliquer, bouger la souris provoquent l'envoi d'événements. Certains événements sont générés, non par l'utilisateur, mais par d'autres applications. Par exemple, si une région cachée de la fenêtre est exposée parce qu'une autre application a été déplacée, le client doit redessiner cette partie et reçoit donc un événement.

Tout comme les requêtes, les événements sont asynchrones. Ils peuvent survenir à tout moment, dans n'importe quel ordre, et il n'y a pas d'accusé de

[10] Comme la taille de la requête doit être un multiple de quatre, quelques octets, dont le contenu est sans importance, peuvent être ajoutés à la fin de la requête.

réception. Le serveur envoyant les événements de son propre chef, l'application cliente ne doit pas continuellement demander au serveur si un événement est survenu. Les événements sont placés dans une file d'attente dans l'ordre où ils arrivent et sont habituellement traités par l'application cliente dans ce même ordre. Une application peut spécifier au serveur quel type d'événement elle souhaite recevoir. Par exemple, une application de dessin *à main levée* peut s'intéresser au déplacement du curseur. Deux applications peuvent vouloir être averties des événements survenant à une même fenêtre; le serveur enverra alors une copie des événements à chacune des applications.

2.8. *Extensions au serveur*

Certains serveurs offrent des fonctionnalités de multimédia, de dessin en trois dimensions, de gestion de périphériques autres que le clavier et la souris, etc. Ces fonctionnalités sont rendues possibles par l'adjonction d'*extensions* au serveur de base. Entre deux versions du serveur, l'ajout de nouvelles fonctionnalités se fait par ce mécanisme plutôt que par une modification dans le serveur même. Cela assure ainsi une compatibilité avec les versions précédentes.

Ce mécanisme permet à des développeurs d'ajouter de nouvelles fonctionnalités au serveur et de rendre ces améliorations disponibles pour les serveurs d'autres vendeurs dans un format standard.

Parmi les extensions livrées avec le serveur X du MIT, notons *MIT-SHM*, un module de gestion de mémoire partagée. Lorsque le serveur et l'application cliente s'exécutent sur le même processeur, il est possible de gagner du temps et de l'espace mémoire en plaçant certaines structures dans une mémoire partagée par le serveur et l'application cliente. Notons également l'extension *SHAPE* qui permet aux fenêtres d'avoir une forme non rectangulaire.

Pratiquement, une extension étend le protocole X par l'ajout de requêtes. La plupart du temps, une librairie est également créée. Elle étend la librairie standard Xlib.

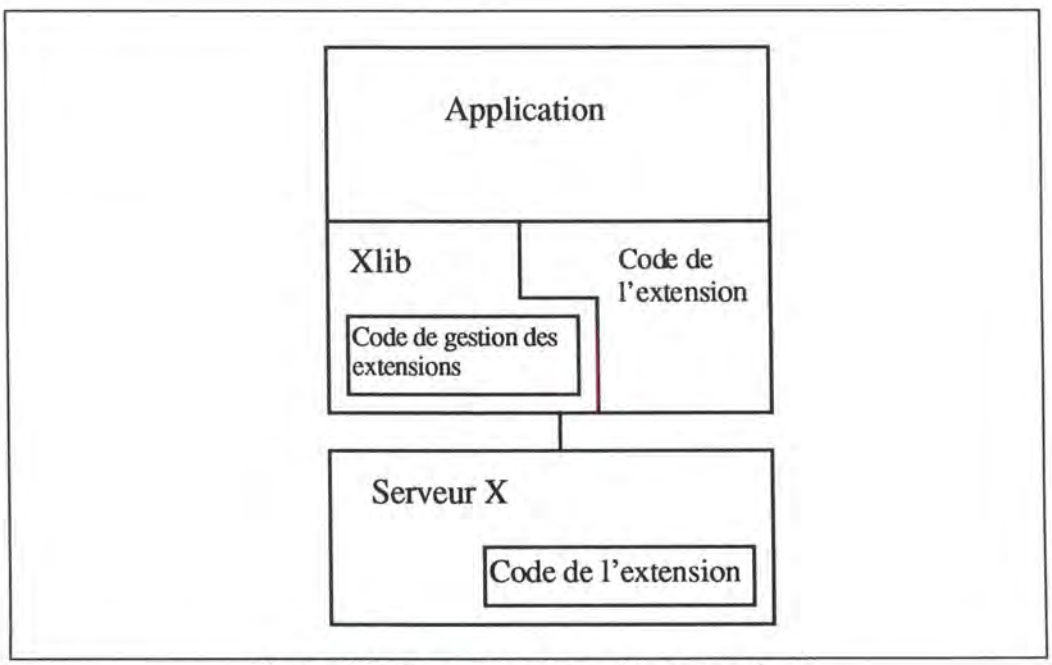

Figure 22 - Le mécanisme des extensions

Lorsque le serveur X reçoit une requête, il extrait le code opératoire majeur, qui est une valeur entière, et l'utilise comme l'index d'un tableau *ProcVector* qui contient les adresses des fonctions traitant chacune un type de requête. Il existe donc une entrée pour la fonction qui traite la requête de tracé

de lignes, une entrée pour la fonction qui traite la requête d'informations sur une police de caractères, etc.

Le principe de l'extension est simple. Le fait d'ajouter une extension revient à ajouter une entrée dans ce tableau. Mais, le tableau étant limité à 256 entrées, dont 128 sont réservées à X, les programmeurs d'extension sont invités à n'utiliser qu'une seule entrée par extension. Dans ce cas, il faut un moyen pour distinguer les différentes requêtes d'une même extension. C'est ainsi qu'est introduit un code opératoire mineur (*minor opcode*) qui peut être utilisé comme deuxième niveau de branchement. Dans ce cas, la fonction pointée par le tableau ProcVector est appelée fonction *dispatch*. Bien sûr, chacun peut implémenter sa fonction dispatch comme il l'entend. Par exemple, elle peut se résumer à une instruction `switch` qui appellera la fonction correspondant au code opératoire mineur (cf. Figure 23); elle peut également appeler une fonction dont l'adresse se trouve dans l'élément d'un second tableau indexé par le code opératoire mineur (cf. Figure 24).

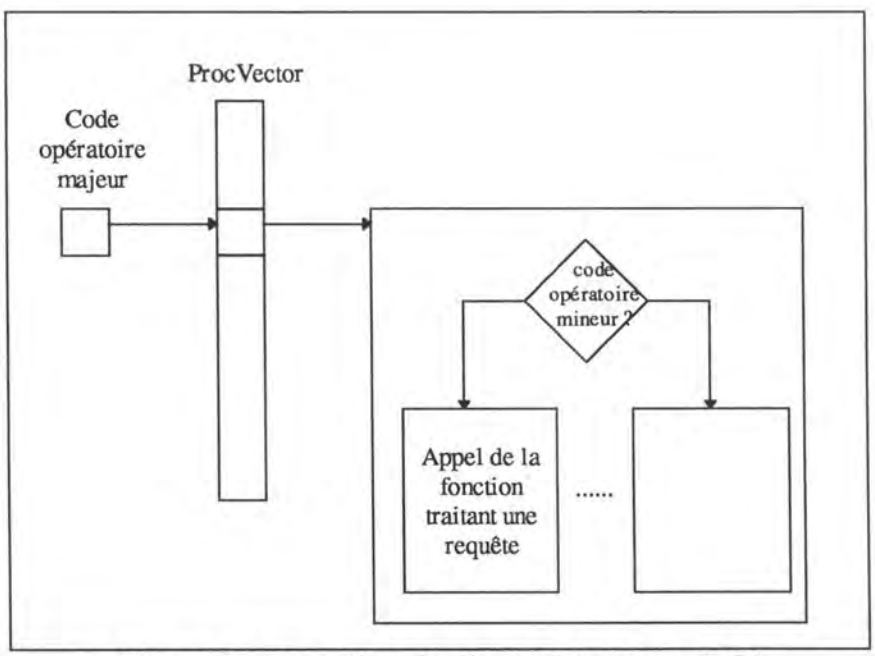

Figure 23 - Fonction de dispatch (avec switch)

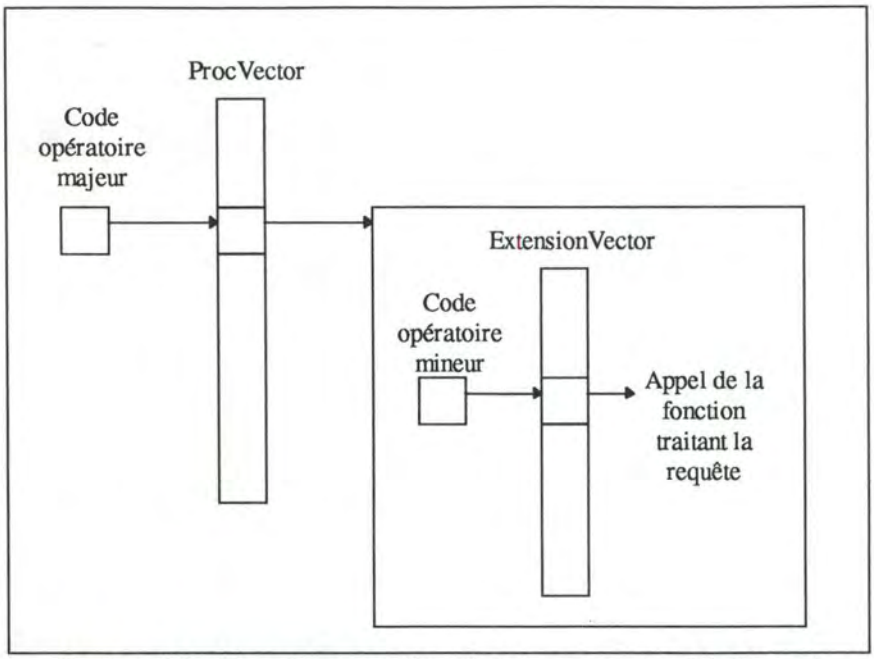

Figure 24 - Fonction de dispatch (avec second tableau)

CHAPITRE 3. LE PROJET PAGE

Les deux premiers chapitres nous ont permis de fixer un certain nombre de notions indispensables à la compréhension des chapitres suivants. Le chapitre 3, quant à lui, est destiné à la présentation du projet PAGE. C'est dans le cadre de ce projet qu'est apparu le problème qui a donné naissance à ce mémoire. Ce problème sera présenté au chapitre 4, et ne sera donc pas abordé dans ce chapitre.

Pour réaliser ce chapitre, nous nous sommes principalement inspiré d'articles émanant du Centre de Recherche Public de Luxembourg[11], un des deux principaux partenaires du projet PAGE.

3.1. Contexte

En 1984, la société SILIS, une société luxembourgeoise de développement de logiciels spécialisée dans le domaine du traitement de texte, développa et commercialisa son premier logiciel : Epistole PC. Ce dernier était un traitement de texte sous DOS orienté caractères. Dès 1987, SILIS, consciente de l'avènement des environnements graphiques, se lança dans le développement d'un nouveau produit destiné à remplacer à terme Epistole PC. Ce nouveau traitement de texte, Interscript, se différenciait principalement de son prédécesseur grâce à son interface utilisateur graphique ainsi qu'à sa propriété de portabilité. En effet, dans un souci d'étendre son marché potentiel, SILIS voulait se doter d'un logiciel portable vers tout type de plate-forme (système ou matérielle). Cette préoccupation a donné naissance au projet PAGE.

---

[11] MOUSEL P. and RETTER P., *La portabilité des applications informatiques dans des environnements graphiques multiples*, CRP-CU report CREDI-A-91-004, Luxembourg (1991).
BARBA F., MOUSEL P., NOGACKI G., RETTER P., *GRAVITI : une boîte à outils virtuelle graphique*, CRP-CU, Luxembourg.
MOUSEL P., NOGACKI G., RETTER P., *Maximum Abstraction as a Path Towards Portability in Multiple Graphical Environments*, CRP-CU report CREDI-R-92-009, Luxembourg.
BARBA F., MOUSEL P., RETTER P., *Application Portability in Multiple Graphical Environments*, CRP-CU report CREDI-R-92-047, Luxembourg.
MOUSEL P., NOGACKI G., RETTER P., *Maximum Abstraction as a Path Towards Portability in Multiple Graphical Environments - Presentation at the IFIP WG2.7 Working Conference*, CRP-CU report CREDI-R-92-055.
DELPERDANGE T., DEMANGE A., GUILLAUME C., MAILLET E., MOUSEL P., POLEUR M., RETTER P., *Design and Implementation of an X Print Server*, CRP-CU, Luxembourg, 1994.
MEINADIER J.P., *L'interface utilisateur - Pour une informatique plus conviviale*, DUNOD, Paris, 1991.
CRP-CU, OFFIS, *Interfaces Utilisateurs Graphiques sous UNIX - Cours de formation au système UNIX*, Luxembourg.

PAGE est l'abréviation de "Portability of Applications in multiple Graphical Environments". Comme son nom l'indique, ce projet s'intéresse à la portabilité des applications dans des environnements graphiques multiples. Pour mener à bien ce projet, la société SILIS travaille en collaboration avec le Centre de Recherche Public - Centre Universitaire (CRP-CU) de Luxembourg. Ce centre de recherche dont la mission est le transfert de technologies, participe à de nombreux projets de Recherche & Développement en collaboration avec des sociétés privées.

Dans le cadre du projet PAGE, le CRP-CU et la société SILIS ont donc uni leurs efforts dans le but de trouver une solution efficace aux problèmes rencontrés lors du portage d'une application informatique d'un environnement graphique vers un autre. Rappelons qu'une application informatique est portable lorsqu'une fois développée dans un environnement matériel et logiciel donné, elle est automatiquement disponible dans un environnement matériel et logiciel différent.

La portabilité, qui est une qualité essentielle d'un logiciel bien conçu, a toujours été tout au long de l'évolution de l'informatique, une source de préoccupation pour les développeurs. Dès le début, lorsque les programmeurs développaient encore leurs applications avec un langage proche de la machine, le langage Assembleur, les premiers problèmes liés à la portabilité apparurent. A cette époque où l'évolution du matériel informatique était très rapide, la décision d'une société de migrer vers un autre type de machine entraînait inévitablement la réécriture des programmes existants dans le langage propre à cette nouvelle plate-forme. Ce problème qui augmentait considérablement les coûts de développement de logiciels a pu être résolu par des langages de programmation dits de hauts niveaux, tels que Fortran, Cobol, Pascal, etc. Ces langages permettaient aux programmeurs de faire abstraction des détails techniques des machines et de se concentrer entièrement sur les problèmes à résoudre. L'utilisation d'un tel langage garantissait ainsi aux programmeurs la portabilité de leurs applications sur n'importe quel type de plate-forme disposant du compilateur adéquat.

Aujourd'hui, l'apparition des environnements graphiques a à nouveau fait surgir des problèmes de portabilité. Avant d'aborder ces problèmes, il nous semble intéressant de rappeler quels sont les avantages de ces environnement et comment ils ont réussi à s'imposer dans le monde de l'informatique.

L'objectif de ces environnements graphiques est de présenter à l'utilisateur une interface consistante, hautement interactive, intuitive et uniforme. Avec les environnements graphiques, on laisse donc de côté les langages de commande difficiles à assimiler au profit d'interfaces utilisateur graphiques, en anglais Graphical User Interfaces (GUI), permettant la manipulation directe. En effet, avec les interfaces graphiques, l'utilisateur agit directement sur les objets affichés à l'écran. Il visualise immédiatement le résultat de ses actions, et peut, s'il le souhaite, effectuer un retour en arrière. Ce qu'il obtient à l'écran par manipulation directe, en préparant un document textuel ou graphique, sera

répercuté tel quel à l'impression; c'est ce qu'on appelle le WYSIWYG (What You See Is What You Get) ou "ce que vous voyez est ce que vous obtenez".

Les interfaces utilisateurs graphiques ont une longue histoire, elles prennent leurs sources dans les années 60 à l'université de Stanford, avec des études sur la partition d'écran, et donc sur le système de fenêtrage, d'une part, et sur l'utilisation de la souris, d'autre part. Elles se développent au PARC[12] de XEROX dans les années 70, avec notamment une contribution fondamentale dans le domaine des langages orientés objet avec Smalltalk, pour donner, au début des années 80, le Star, terminal bureautique de XEROX. Les concepts ont été repris par Apple qui annonçait en 1983 le micro-ordinateur bureautique LISA. Star et LISA, qui ne bénéficiaient pas à l'époque d'une technologie suffisamment avancée, restaient trop coûteux pour connaître un grand développement. Avec l'apparition des microprocesseurs 32 bits de Motorola (68000), Apple a pu sortir en 1984 le Macintosh. Ce dernier, grâce à son interface utilisateur graphique et sa convivialité légendaire, s'est imposé dans tous les secteurs professionnels. Cinq ans après, le passage des compatibles PC aux microprocesseurs 32 bits d'Intel (386 et 486) a permis à ces machines de bénéficier de logiciels d'interface utilisateur du même type avec Windows et Presentation Manager. Parallèlement, les universités américaines et les industriels des stations de travail scientifiques, Apollo, SUN, DEC, HP, par la suite suivis par NeXT, développaient les interfaces utilisateurs du monde UNIX, aboutissant à des produits tels que X-Window, Motif, Open Look, NextStep qui apparaissent comme des standards du marché.

Comme nous venons de le voir, à l'heure actuelle, des interfaces utilisateurs graphiques sont disponibles sur pratiquement toutes les catégories de matériel. Avec ce type d'interfaçage, les utilisateurs d'aujourd'hui sont équipés, en plus de l'habituel clavier, d'une souris et d'un écran graphique. Les applications ont également évolué au niveau du type de dialogue. Avant, le dialogue était entièrement contrôlé par l'ordinateur, qui présentait à l'utilisateur des écrans successifs. Aujourd'hui, on est passé à un type de dialogue contrôlé par les actions de l'utilisateur. Ces actions sont traduites sous forme d'événements auxquels l'ordinateur doit répondre.

La qualité principale d'une interface utilisateur graphique est sa consistance. Une interface est dite consistante ou cohérente ou encore homogène si elle est prévisible quel que soit le contexte ou l'application, c'est-à-dire si une action donnée a toujours les mêmes conséquences[13]. Une interface consistante facilite l'apprentissage des utilisateurs face à une application nouvelle puisqu'elle leur permet de travailler dans un environnement déjà connu et ainsi de réutiliser leurs acquis. Cette homogénéité dans l'interface est rendue possible, d'une part, par la présence dans chaque environnement graphique de boîtes à outils; celles-ci étant des ensembles de fonctions et de structures de données prédéfinies et mises à la disposition des programmeurs afin de leur éviter de

[12] Palo Alto Research Center
[13] MEINADIER J. P., *L'interface utilisateur - Pour une informatique plus conviviale*, DUNOD, Paris, 1991, page 58.

développer leurs applications à partir de zéro. Et d'autre part, de lignes de conduite destinées aux développeurs qui énoncent les grands principes d'utilisation des objets de la boîte à outils au sein des différentes applications.

Revenons maintenant sur les problèmes de portabilité que l'on rencontre lorsque l'on désire développer une application dans des environnements graphiques multiples.

3.2. La portabilité des applications dans des environnements graphiques multiples

Lorsqu'un programmeur développe une application dans un environnement graphique, il utilise les services offerts par la boîte à outils. Chaque environnement graphique dispose de sa propre boîte à outils. Actuellement, on assiste à une standardisation au niveau sémantique des interfaces utilisateurs, c'est-à-dire que l'on retrouve les mêmes fonctionnalités dans chacun des environnements. Mais, les interfaces de programmation des différentes boîtes à outils sont loin d'être standardisées ce qui entraîne de sérieux problèmes de portabilité. En effet, pour porter une application d'un environnement graphique vers un autre, il faudrait pouvoir isoler dans l'application les parties dépendantes et indépendantes du système. Une fois que ces parties auraient été identifiées, il suffirait de réécrire les parties dépendantes du système pour que l'application soit portable sur un autre système. Malheureusement, les parties dépendantes du système, dans les applications destinées à s'exécuter dans des environnements graphiques sont de loin les plus importantes. En cause, la part importante de code dont la mission est de gérer l'interface utilisateur graphique de l'application. On estime en moyenne à 50% cette partie de l'application dans les environnements graphiques. Pour gérer l'interface, les applications font appel aux services d'une boîte à outils, ce qui a pour conséquence de rendre pratiquement impossible l'identification d'un noyau fonctionnel indépendant du système. Porter une application dans des environnements graphiques multiples n'est donc pas chose aisée. Une solution pour pouvoir disposer d'une application dans plusieurs environnements graphiques est de réécrire cette application pour chaque environnement envisagé. Cette solution n'est pas économiquement viable puisqu'elle risque d'engendrer des temps de développement exorbitants ainsi que d'énormes problèmes de maintenance. De plus, avec cette solution, peut-on encore parler d'application portable ?

Il y a, cependant, différents moyens qui permettent d'exécuter une application dans plusieurs environnements graphiques :

Un émulateur

Le premier de ces moyens consiste à utiliser un logiciel particulier appelé émulateur. Ce programme a la particularité de transformer le système sur lequel il s'exécute en un autre système ayant une architecture matérielle et un système d'exploitation différents. Il permet alors d'exécuter en mode natif toutes les applications disponibles dans le système émulé. Comme émulateur sur le marché, on trouve notamment le programme SoftPC de Insignia qui permet aux utilisateurs de Macintosh ou de stations de travail UNIX de faire tourner des applications natives sous DOS ou Windows. Cette solution est séduisante

puisqu'elle permet de porter immédiatement toutes les applications développées dans un environnement vers un autre, à condition que ce dernier dispose de l'émulateur adéquat. Malheureusement, l'utilisation d'un tel programme pose inévitablement des problèmes de performance et d'adaptabilité à un environnement peu familier. De plus, avec cette solution, les développeurs d'applications dépendent fortement de l'existence d'émulateurs émulant leur propre système sur d'autres environnements. A moins, bien sûr, qu'ils ne développent eux-mêmes leurs propres émulateurs pour chaque environnement ciblé. Mais concevoir un émulateur est une tâche assez complexe qui ne peut être prise en charge par les développeurs d'applications pour chaque plate-forme.

Une interface entre deux systèmes

Pour pallier les problèmes de performance présents lorsqu'on utilise un émulateur, une solution consiste à utiliser une interface qui va répondre aux demandes de service des applications conçues pour un environnement graphique mais s'exécutant sur un autre. Remarquons que cette solution n'est possible que sur des systèmes ayant une architecture matérielle identique. Comme dans la solution précédente, l'ensemble des applications développées sur un système sont automatiquement disponibles sur un autre si ce dernier possède l'interface adaptée. En adoptant cette façon de procéder, les développeurs éliminent les problèmes de performance rencontrés en utilisant un émulateur, mais sont toujours confrontés, s'ils désirent une totale indépendance, au travail complexe de conceptions de l'interface pour chacune des plates-formes envisagées.

Un exemple d'interface de ce type est le Windows Application Binary Interface (WABI) de SunSoft qui permet aux utilisateurs de PC d'exécuter des applications Windows en mode natif.

Les deux solutions proposées jusqu'à présent permettent aux programmeurs de porter leurs applications en ne les développant qu'une seule fois dans leur environnement privilégié. La portabilité est cependant limitée, d'une part, aux systèmes pour lesquels il existe un émulateur, et d'autre part, à ceux possédant en plus de la même architecture matérielle, d'une interface fournissant les services attendus des applications déportées. L'inconvénient majeur de ces deux solutions réside dans le manque d'indépendance des développeurs qui sont liés fortement à l'existence de programmes spécialisés et difficiles à implémenter pour la portabilité de leurs applications.

Un langage de programmation

Une solution pour porter une application d'un environnement graphique vers un autre serait d'utiliser des langages de programmation qui intègrent pleinement les caractéristiques graphiques de ces environnements. Un exemple de langage de ce type est le langage Smalltalk. Ce langage orienté objet a été le premier environnement de programmation d'interfaces utilisateur graphiques. Il trouve son origine au PARC de XEROX lors d'études sur les interfaces utilisateur. Les programmeurs utilisant ce langage pour développer leurs applications ont l'assurance que ces dernières sont automatiquement disponibles sur toutes les plates-formes disposant d'un environnement Smalltalk. Cependant, les développeurs sont souvent réticents à l'idée de délaisser un langage de programmation qu'ils maîtrisent et dont il existe des compilateurs sur un grand nombre de machines, au profit d'un autre peu connu et qui bien souvent ne respecte pas le *look and feel* d'origine des environnements graphiques.

Une boîte à outils virtuelle

La dernière des solutions met en jeu le concept de boîte à outils virtuelle. L'idée d'implémenter une telle boîte à outils provient de la façon dont le système d'exploitation s'acquitte de sa tâche vis-à-vis des différentes applications. Le système d'exploitation, en présentant aux applications une machine virtuelle, permet à celles-ci de ne pas avoir à se préoccuper des détails techniques de l'ordinateur. De la même façon, une boîte à outils virtuelle pour les environnements graphiques constituerait une couche logicielle entre les applications et les différentes boîtes à outils de ces environnements. Cette boîte à outils virtuelle présenterait aux applications une interface de programmation unique et cacherait ainsi les caractéristiques des boîtes à outils propres à chaque environnement (cf. Figure 25). Ainsi une application développée avec cette boîte à outils virtuelle, c'est-à-dire faisant exclusivement appel aux services de cette boîte, serait automatiquement disponible dans tous les environnements graphiques chapeautés par cette nouvelle couche logicielle.

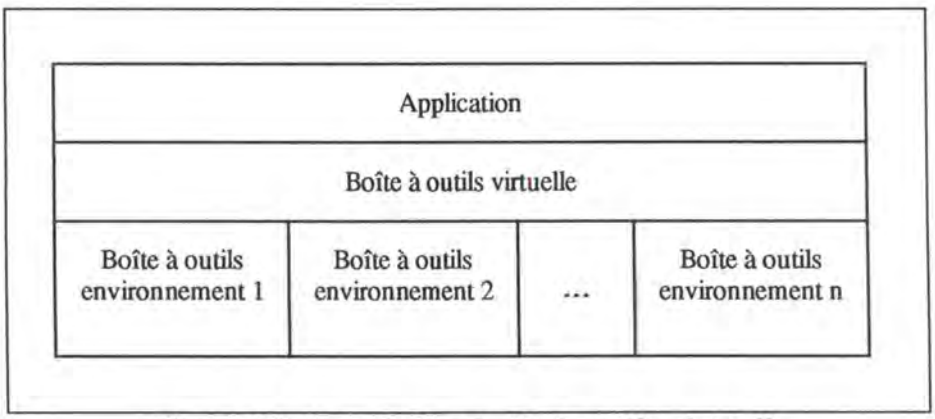

Figure 25 - La solution boîte à outils virtuelle

Les avantages que cette solution procure sont multiples. Premièrement, il suffit de porter cette boîte à outils virtuelle dans un environnement donné pour que toutes les applications développées à partir de cette boîte à outils soient disponibles dans cet environnement. Deuxièmement, la boîte à outils virtuelle peut être écrite dans un langage de programmation traditionnel, ce qui ne met pas en péril les investissements avancés dans des applications précédentes. Troisièmement, l'utilisation d'une boîte à outils virtuelle permet aux développeurs de viser un marché potentiel élargi pour la commercialisation de leurs produits, vu qu'une fois qu'une application est développée, elle est disponible immédiatement dans tous les environnements sur lesquels la boîte à outils a été portée. Quatrièmement, l'élargissement du marché potentiel permet de diminuer le risque commercial encouru par des sociétés auparavant limitées à un seul environnement graphique. Et enfin, cinquièmement, en utilisant une boîte à outils virtuelle, on limite bien entendu les coûts de maintenance de logiciels, vu qu'une fois une version du programme mise à jour, elle est automatiquement disponible dans tous les environnements pris en charge par la boîte à outils.

Naturellement, lorsque l'on désire faire appel à un service d'une boîte à outils d'un environnement particulier, on doit passer par le nouvel intermédiaire qu'est la boîte à outils virtuelle. Cela a pour effet de pénaliser le temps de réponse de l'application. Toutefois, cet inconvénient est minime comparé aux avantages que procurent les services de la boîte à outils virtuelle, notamment la portabilité des applications dans plusieurs environnements graphiques.

3.3. La boîte à outils virtuelle GRAVITI

Dans le cadre du projet PAGE, qui visait à améliorer la portabilité des applications dans des environnements graphiques multiples, l'approche qui a été retenue est celle de la boîte à outils virtuelle.

Le CRP-CU et la société SILIS ont décidé de développer leur propre boîte à outils virtuelle qu'ils appellent GRAVITI pour GRAphical VIrtual ToolkIt ainsi que des logiciels complémentaires tels que des compilateurs de ressources virtuelles et des compilateurs d'aide. GRAVITI permet d'offrir aux développeurs d'applications une interface de programmation unique pour des environnements graphiques différents. Ainsi une application développée avec GRAVITI est automatiquement disponible pour tous les environnements sur lesquels cette boîte à outils a été portée.

Dès le départ plusieurs décisions importantes ont été prises. Premièrement dans le choix du langage de programmation pour le développement de GRAVITI. Les concepteurs ont porté leurs préférences sur le langage C, en raison de sa standardisation au niveau international ainsi que de sa disponibilité sur pratiquement toutes les plates-formes. La deuxième décision portait sur le choix des environnements graphiques sur lesquels GRAVITI serait portée. Les quatre environnements graphiques les plus populaires dans le monde de l'informatique, à savoir Microsoft Windows, OS/2 Presentation Manager, Macintosh et OSF Motif, ont été choisis, ce qui permet aux développeurs d'applications utilisant GRAVITI de disposer d'un large marché potentiel.

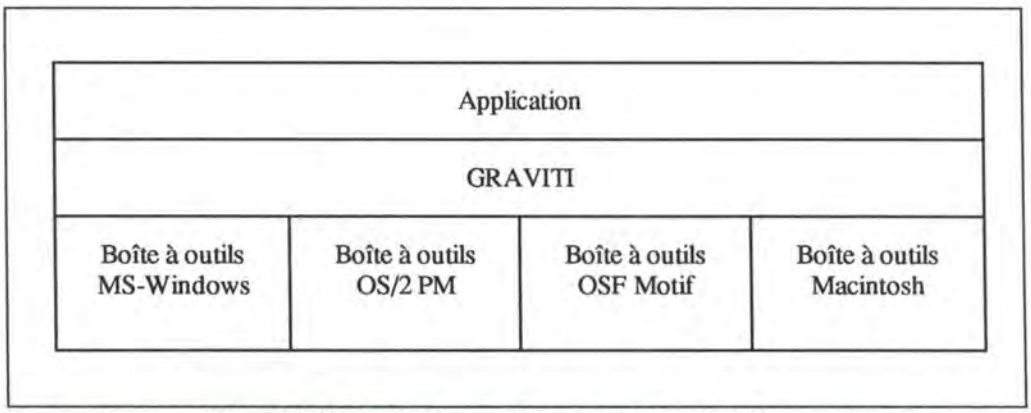

Figure 26 - GRAVITI est portée sur quatre environnements

Troisièmement, le CRP-CU et SILIS ont décidé, dans le but de conserver la consistance des interfaces des différents environnements, de respecter le *look and feel* d'origine de chaque environnement graphique. Ainsi une application développée avec GRAVITI et compilée sur un système Macintosh, aura la même apparence et se comportera de la même façon que toutes les autres applications développées pour Macintosh. Il en est de même pour les applications des trois

autres environnements graphiques. Enfin, il a fallu décider quelles seraient les fonctionnalités offertes par la boîte à outils virtuelle. Une façon de procéder eut été d'énumérer les fonctionnalités de chaque environnement graphique et de se limiter à l'implémentation des caractéristiques communes aux quatre environnements. Cette approche qui peut être qualifiée de minimaliste présente un inconvénient majeur; cet ensemble de fonctionnalités communes peut être vide. A l'opposé, on trouve l'approche maximaliste qui consiste à implémenter, dans la boîte à outils virtuelle, toutes les caractéristiques de chaque environnement. Cette solution est la plus ambitieuse, mais est difficilement implémentable puisque des fonctionnalités présentes dans certains environnements sont impossibles à implémenter dans d'autres.

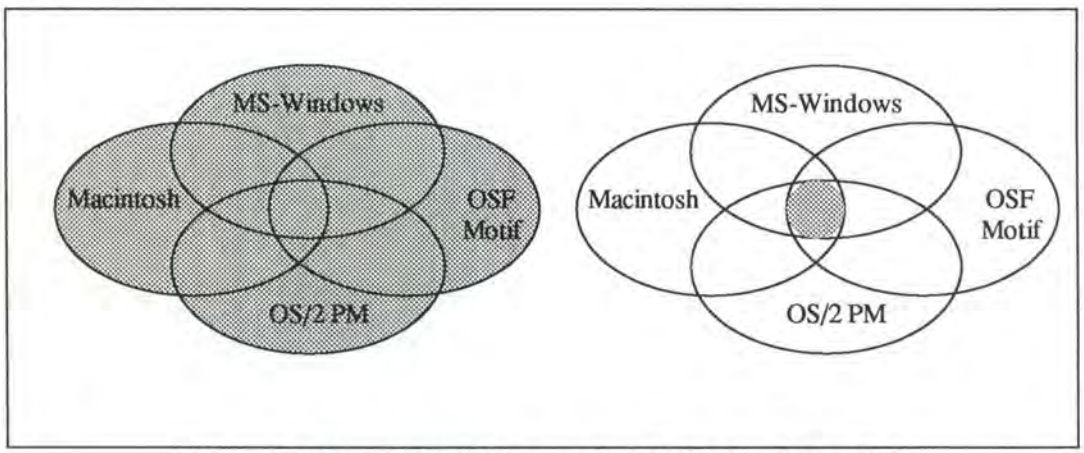

Figure 27 - Les approches maximaliste et minimaliste

Une solution intermédiaire serait de déterminer un environnement virtuel suffisamment général pour permettre le développement d'une gamme étendue d'applications. Ainsi, on retrouverait dans GRAVITI, non pas l'union, ni l'intersection de caractéristiques de chaque environnement mais un ensemble acceptable de fonctionnalités permettant l'implémentation d'un grand nombre de programmes classiques. C'est cette solution qui a été retenue dans le cadre du projet PAGE.

Nous savons maintenant que GRAVITI offre aux développeurs d'applications des services raisonnables qui sont des abstractions des fonctionnalités disponibles dans chaque environnement. Pour nous permettre de comprendre la nature de ces services, nous allons nous intéresser à l'architecture interne de GRAVITI. Comme on le voit sur la Figure 28, GRAVITI est composée de sept modules spécialisés chacun dans un domaine particulier.

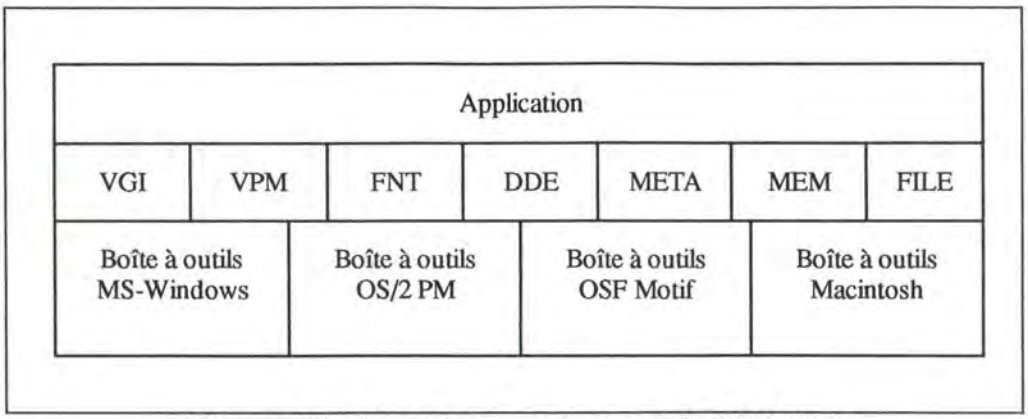

Figure 28 - L'architecture interne de GRAVITI

Le module Virtual Graphical Interface (VGI) permet les opérations graphiques de base telles que le dessin de lignes, de textes et d'objets de toutes sortes. Le Virtual Presentation Manager (VPM) s'occupe de la gestion des objets faisant partie de l'interface graphique, du presse papier et des services d'impression. Le module dédié aux polices de caractères est le module Font (FNT) et celui s'occupant des liens dynamiques entre les documents est le Dynamic Data Exchange (DDE). Ensuite, on trouve le module Metafile (META) qui traite le graphisme vectoriel dépendant du système. Les deux derniers modules soutiennent, d'une part, la gestion dynamique de la mémoire, le Memory Management Module (MEM), et d'autre part, la gestion des fichiers portables avec le File Access Module (FILE).

Appuyons les affirmations du point 3.2. concernant les avantages et les gains de productivité que procurent l'utilisation d'une boîte à outils virtuelle par quelques chiffres.

Modules	Commun	OS/2	Windows	OSF Motif	Macintosh	Total
VGI	1553 12,3%	1725 13,7%	2464 19,5%	2708 21,5%	4169 33%	12619
VPM	6395 13,6%	6268 13,4%	7379 15,7%	9156 19,5%	17731 37,8%	46929
FNT	4539 60%	601 8%	1001 13,2%	824 10,9%	595 7,9%	7560
DDE	4502 94,2%	140 2,9%	139 2,9%			4781
META	1616 49,4%	440 13,4%	646 19,7%	42 1,3%	529 16,2%	3273
MEM	3793 100%					3793
FILE	2279 41,8%	468 8,6%	224 4,1%	415 7,6%	2066 37,9%	5452
GRAVITI	24677 29,3%	9642 11,4%	11853 14%	13145 15,6%	25090 29,7%	84407

Tableau 1 - Détail du code source de GRAVITI

Le tableau 1 nous renseigne sur le nombre de lignes de code contenues dans chacun des sept modules de GRAVITI pour chaque environnement graphique. Il est incomplet dans la mesure où le modules DDE n'est pas encore implémenté dans les environnements Macintosh et OSF Motif. Toutefois, les données concernant ces implémentations futures n'influenceront pas de manière significative les résultats obtenus jusqu'à maintenant.

Lorsqu'on analyse le tableau 1, on remarque que pour quatre des sept modules de GRAVITI, plus de 50% de l'effort de programmation au sein de ces modules est commun aux quatre environnements. Le module de gestion de la mémoire (MEM) ne contient même aucune ligne de code dépendante d'un système particulier. La dernière colonne indique la taille de chaque module tous environnements confondus. La dernière ligne, quant à elle, nous rappelle la part occupée par le code propre à chaque environnement par rapport à la taille totale de GRAVITI. Cette dernière ligne est plus importante qu'il n'y paraît puisqu'elle nous permet d'affirmer qu'en moyenne porter GRAVITI dans un nouvel environnement graphique nécessite la réécriture de plus ou moins 15000 lignes de code. Cela signifie qu'un effort de programmation de plus ou moins 15000 lignes de code permettrait à l'ensemble des applications développées avec GRAVITI, c'est-à-dire actuellement au seul programme Interscript, d'être disponible dans un environnement graphique supplémentaire.

Dans le but de démontrer que la boîte à outils virtuelle GRAVITI permet de concevoir des applications professionnelles de qualité, la société SILIS et le CRP-CU ont décidé d'implémenter le nouveau traitement de texte Interscript à l'aide de ses services.

3.4. Interscript : une application bureautique professionnelle développée à l'aide de GRAVITI

Le traitement de texte Interscript est la première application professionnelle développée en utilisant uniquement les services offerts par la boîte à outils virtuelle du projet PAGE. Il n'a donc été développé qu'une seule fois avec GRAVITI pour être automatiquement disponible dans tous les environnements graphiques sur lesquels la boîte à outils virtuelle a déjà été portée. En l'occurrence OS/2 Presentation Manager et Microsoft Windows. Certains modules de GRAVITI sont toujours en cours de développement pour les deux autres environnements.

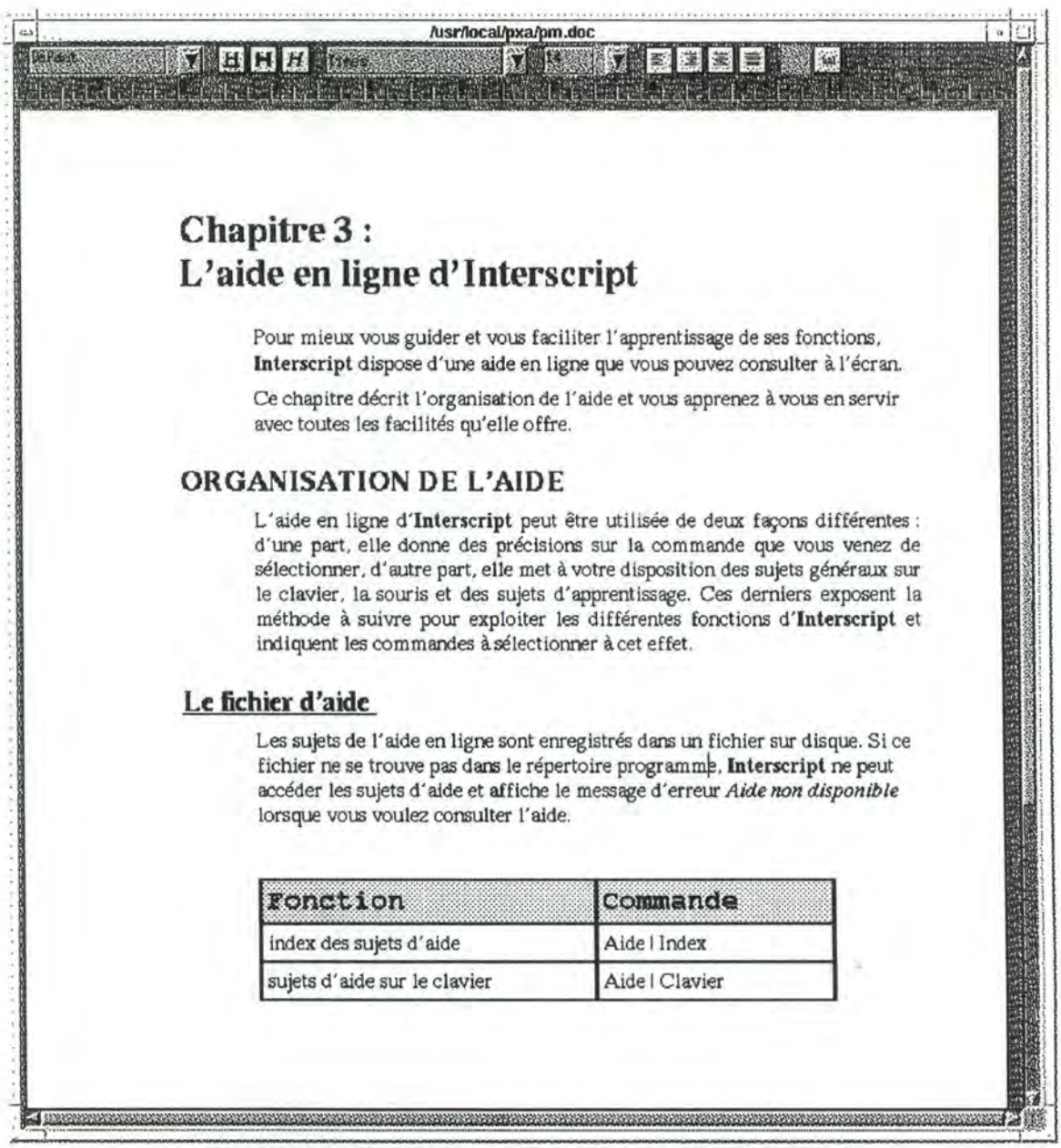

Figure 29 - Copie écran d'un document Interscript

Interscript est un logiciel de traitement de texte complet destiné à être utilisé dans les environnements graphiques. Il propose aux utilisateurs tous les services généralement offerts par ce type de programme; à savoir toutes les opérations sur les documents, l'alignement de paragraphes, l'affichage de textes stylisés, des fonctions d'édition (couper/coller), une aide en ligne, la présence d'un dictionnaire, etc.

L'objectif final des partenaires du projet PAGE, qui était de montrer qu'il était possible de porter une application dans des environnements graphiques multiples, a donc été pleinement atteint depuis que la société SILIS commercialise Interscript dans deux des environnements cibles.

CHAPITRE 4. PROBLÈME D'IMPRESSION SOUS X

Nous l'avons vu dans le chapitre précédent, un des modules de la boîte à outils virtuelle GRAVITI, le module VPM, s'occupe en plus de la gestion des objets de l'interface et du presse-papiers, d'offrir aux applications clientes un service d'impression virtuel.

Lorsqu'une application développée avec GRAVITI veut adresser l'imprimante, elle a donc recours exclusivement à des fonctions de ce module. Une fois appelées, ces fonctions en appellent d'autres qui sont spécifiques à l'environnement graphique sur lequel l'application a été compilée. Naturellement, pour que cela soit possible, il faut que les services d'impression soient disponibles dans chaque environnement graphique envisagé.

Malheureusement, de tels services d'impression peuvent parfois faire défaut: ils sont notamment absents dans l'environnement X-Window. Cela a posé de sérieux problèmes aux concepteurs de GRAVITI qui devaient proposer une abstraction de service d'impression qui n'existait pas dans un des quatre environnements.

La suite de ce chapitre nous permettra, d'une part, de présenter la façon dont l'impression est effectuée dans deux environnements graphiques très connus, Macintosh et Microsoft Windows, et d'autre part, de nous intéresser à la façon dont cela se passe pour le système X-Window.

4.1. L'impression sous Macintosh

Imprimer sous Macintosh se fait plus ou moins de la même manière qu'afficher à l'écran. Dans les deux cas, le programmeur utilise les services d'un module appelé *QuickDraw* qui permet toutes les opérations graphiques de base. Celles-ci sont toujours appliquées à un *GrafPort* qui peut être comparé au Graphic Context (GC) du système X-Window. Un GrafPort est une structure de données plus ou moins complexe reprenant des informations sur des paramètres graphiques. On y trouve notamment un champ particulier permettant d'identifier le périphérique auquel ce GrafPort est associé, un écran ou une imprimante par exemple.

Les applications Macintosh manipulent au minimum un GrafPort associé à l'écran, mais généralement, elles en utilisent plusieurs vu qu'une fenêtre est un GrafPort particulier. A chaque fois qu'une application crée une fenêtre, elle crée donc un GrafPort qu'elle utilisera pour afficher des résultats dans cette fenêtre.

Mais les opérations graphiques du module QuickDraw, qui permettent d'afficher et d'imprimer, ne peuvent être appliquées qu'à un GrafPort particulier appelé GrafPort courant. Ainsi, avant de pouvoir afficher des résultats dans une fenêtre, une application doit d'abord créer cette fenêtre, donc créer un GrafPort. Ensuite, elle sauvegarde le GrafPort courant et le remplace par celui de la fenêtre. A partir de ce moment, toutes les opérations graphiques de l'application seront appliquées à cette fenêtre. Enfin, lorsque l'affichage dans cette fenêtre a pris fin, l'application restaure l'ancien GrafPort courant sauvegardé et efface le GrafPort de la fenêtre si celle-ci n'est plus utilisée par la suite.

Une application Macintosh suit une procédure identique pour imprimer; la seule différence réside dans la création du GrafPort. Une application ne peut imprimer dans cet environnement que si elle dispose d'un GrafPort particulier appelé GrafPort d'impression.

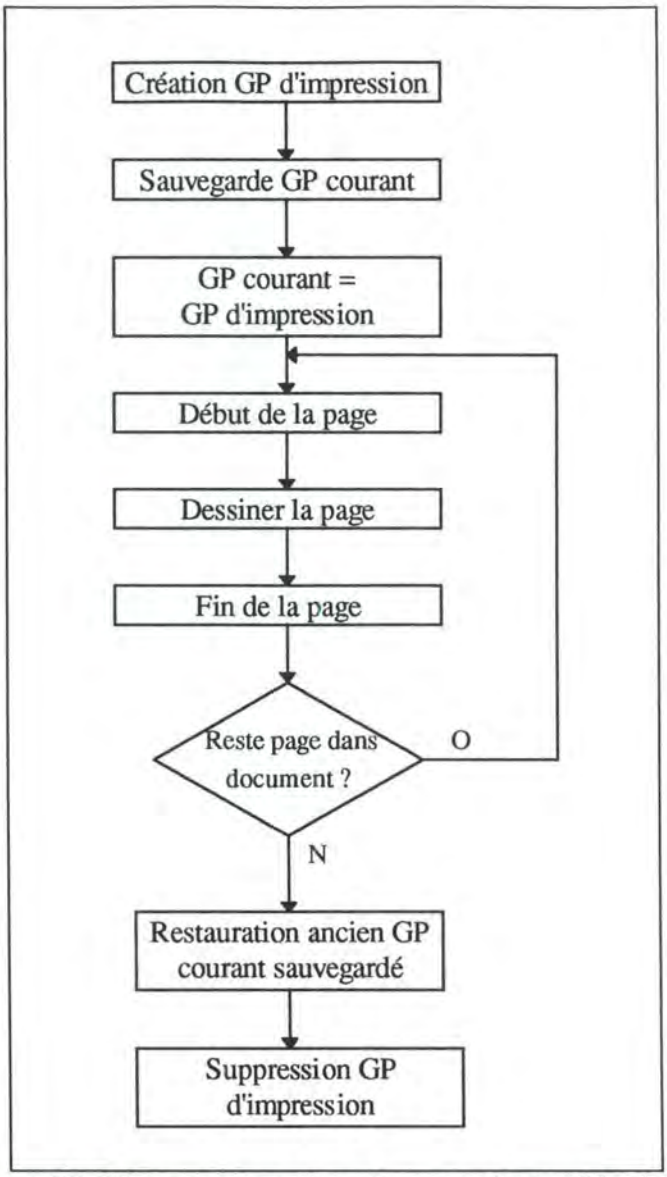

Figure 30 - L'impression d'un document sous Macintosh

La Figure 30 décrit la procédure que suit toute application Macintosh lorsqu'elle désire adresser l'imprimante. Avant toute chose, elle demande la création d'un GrafPort d'impression qui va remplacer le GrafPort courant une fois que ce dernier aura été sauvegardé par l'application. Ensuite, l'application fait appel à une fonction du QuickDraw spécifique aux documents imprimés. Cette fonction a pour but de marquer le début d'une nouvelle page au sein du document. Elle ne peut être appliquée qu'à un document imprimé, en raison de son format particulier. En effet, contrairement aux fenêtres, un document imprimé peut être composé de plusieurs pages, d'où la nécessité de disposer de fonctions permettant de gérer cette caractéristique propre à l'impression. On distingue deux fonctions de ce style dans le module QuickDraw, une pour marquer le début d'une nouvelle page, et une autre pour indiquer la fin d'une page. Une fois que l'application dispose d'une nouvelle page, elle utilise l'ensemble des procédures graphiques pour dessiner son contenu. Cette opération se termine par l'appel de la fonction de fin de page. Si le document imprimé est

composé de plusieurs pages, les opérations d'ouverture, de dessin et de fermeture d'une page seront répétées autant de fois qu'il y a de pages dans le document. Lorsque la dernière page du document aura été traitée, il ne restera plus à l'application qu'à restaurer l'ancien GrafPort courant préalablement sauvegardé, et à effacer le GrafPort d'impression si ce dernier n'est plus utilisé par la suite.

4.2. L'impression sous Microsoft Windows

L'impression sous Microsoft Windows suit le même schéma que celui de l'environnement Macintosh. L'affichage des résultats dans une fenêtre à l'écran, tout comme l'impression sur un document imprimé, ne sont possibles pour une application qu'à la condition que celle-ci dispose d'un *device context*, respectivement, pour l'écran et pour l'imprimante.

Les procédures graphiques pour adresser l'écran ou l'imprimante sont identiques. Elles prennent en argument un device context qui détermine l'endroit où elles seront appliquées.

Mais comme sous Macintosh, certaines routines graphiques sont exclusivement réservées à l'impression. En effet, contrairement aux fenêtres affichées à l'écran, les documents imprimés sont composés de plusieurs pages, ce qui nécessite un traitement particulier et donc des opérations spécifiques. Sous Microsoft Windows, ces dernières sont au nombre de trois : une première permet de commencer un nouveau document imprimé, une autre permet de marquer la fin de ce même document, et enfin, la dernière s'occupe de la terminaison d'une page au sein d'un document imprimé.

La Figure 31 nous présente les différentes étapes suivies par une application Microsoft Windows pour imprimer.

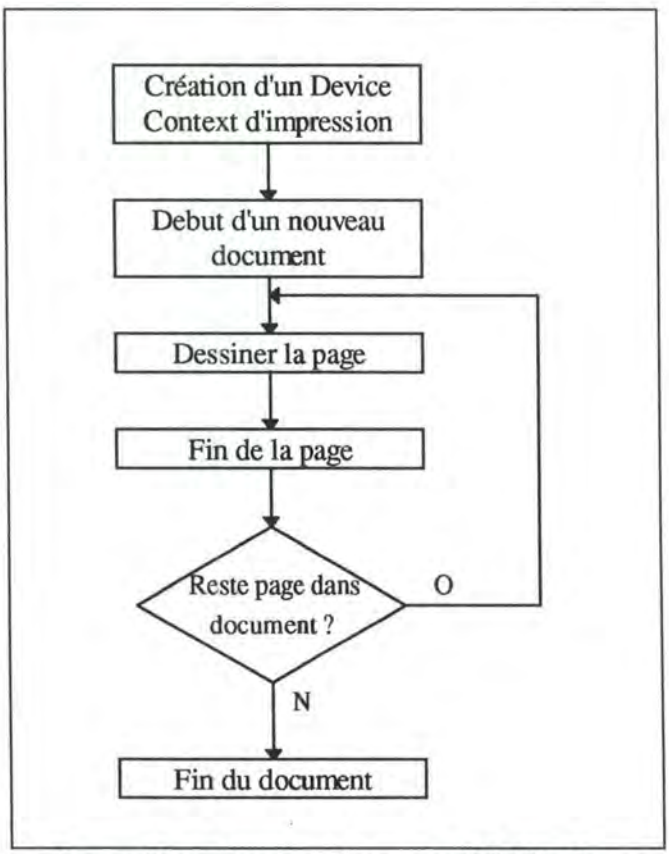

Figure 31 - L'impression sous MS-Windows

Dans l'environnement Microsoft Windows, l'impression est donc toujours précédée de la création d'un device context d'impression. Ensuite, l'application doit spécifier le début du document imprimé ce qui met à sa disposition une page sur laquelle elle peut dessiner à l'aide des routines graphiques. La fin de la page est marquée par l'appel d'une fonction spéciale de terminaison de page. Cette procédure est répétée autant de fois qu'il y a de pages dans le document imprimé. L'impression du document se termine en utilisant l'opération spécifique à la fin d'un document imprimé.

Les deux environnements présentés jusqu'ici ont en commun une caractéristique primordiale pour la productivité des programmeurs, à savoir l'utilisation des mêmes fonctions graphiques pour adresser l'écran ou l'imprimante. Cette propriété permet aux développeurs d'applications pour Microsoft Windows ou pour Macintosh de n'écrire qu'une seule fois les procédures graphiques pour dessiner dans une fenêtre, ou dans un document imprimé. D'où un gain de temps et une diminution des lignes de code du côté de l'application.

Un programme écrit pour l'un de ces environnements et ayant appelé x fonctions graphiques pour tracer un dessin à l'écran appellera donc les x mêmes fonctions graphiques pour dessiner un dessin identique sur un document imprimé.

De plus, la génération de code propre à l'imprimante est prise en charge par le système dans les deux environnements, et est donc transparente pour l'application. Les programmes développés pour ces environnements sont donc totalement indépendants du type de périphérique d'impression utilisé.

4.3. L'impression sous X

Les services d'impression sous X sont réduits à leurs plus simples expressions, c'est-à-dire qu'ils n'existent pas. Pour disposer de tels services, les développeurs d'applications sous X doivent donc trouver une solution intermédiaire. Le chapitre 5 est consacré entièrement à la présentation de ces solutions.

CHAPITRE 5. QUELQUES SOLUTIONS

Pour pallier ce manque de service d'impression dans l'environnement X, il existe différentes solutions. Nous allons en présenter cinq dans ce chapitre. La sixième solution, celle choisie au sein du CRP-CU, sera présentée dans le chapitre suivant.

5.1. La copie écran[14]

La solution de la copie écran (c'est-à-dire la recopie de ce qui est affiché vers l'imprimante) est la solution la plus simple, du moins, pour le programmeur. Sous Unix, cette tâche pourrait être réalisée par:

```
xwd | xpr -device ps | lpr
```

`xwd` produit une réprésentation pixel par pixel de la fenêtre choisie. Cette représentation dépend directement de la résolution offerte par l'écran (habituellement 75 ou 100 dpi[15]). `xpr -device ps` transforme la représentation de la fenêtre en code PostScript. Finalement, `lpr` envoie le code PostScript à l'imprimante.

Si cette solution est la plus simple, c'est aussi celle qui présente le résultat le plus médiocre. En effet, cette méthode présente plusieurs inconvénients. D'abord, l'imprimante offrant couramment des résolutions de 300 ou 600 dpi, l'image de la fenêtre n'est pas nette car, à l'écran où il fallait 100 points pour remplir un pouce, il en faudra 300 sur l'imprimante. Ce changement de résolution a pour conséquence un effet d'escalier, particulièrement visible dans les polices de caractères. Ensuite, l'utilisateur doit intervenir lui-même pour effectuer cette impression. De plus, on ne peut imprimer que ce qui est affichable à l'écran. Si une image ne tient pas dans un seul écran (et que la fenêtre offre par exemple la possibilité de faire défiler cette image via un ascenseur), seule la partie visible de cette image pourra être imprimée. Et, à l'inverse, toute la fenêtre sera imprimée, y compris les ascenseurs et la barre de menu. Enfin, ce processus consomme du temps et de l'espace (l'image pixel par pixel doit être stockée provisoirement dans un fichier intermédiaire).

14 RAVES W., *A PostScript X Server*, The X Resource, n°1, 1992, p. 34
15 DPI = *Dots per inch* ou points par pouce.

Ce procédé ne peut donc être utilisé pour produire des documents imprimés de bonne présentation. Une application doit donc offrir des possibilités d'impression.

5.2. Un générateur de rapport au sein de l'application[16]

Le programmeur peut écrire une application cliente contenant du code de génération d'un langage de définition de page (PDL; PostScript, PCL, HPGL, par exemple) spécifique à cette application (cf. Figure 32). On imagine, par exemple, un logiciel de reporting dont le code source contient une fonction qui lit un fichier de données et qui envoie celles-ci à l'imprimante sous forme d'un tableau (une facture, par exemple).

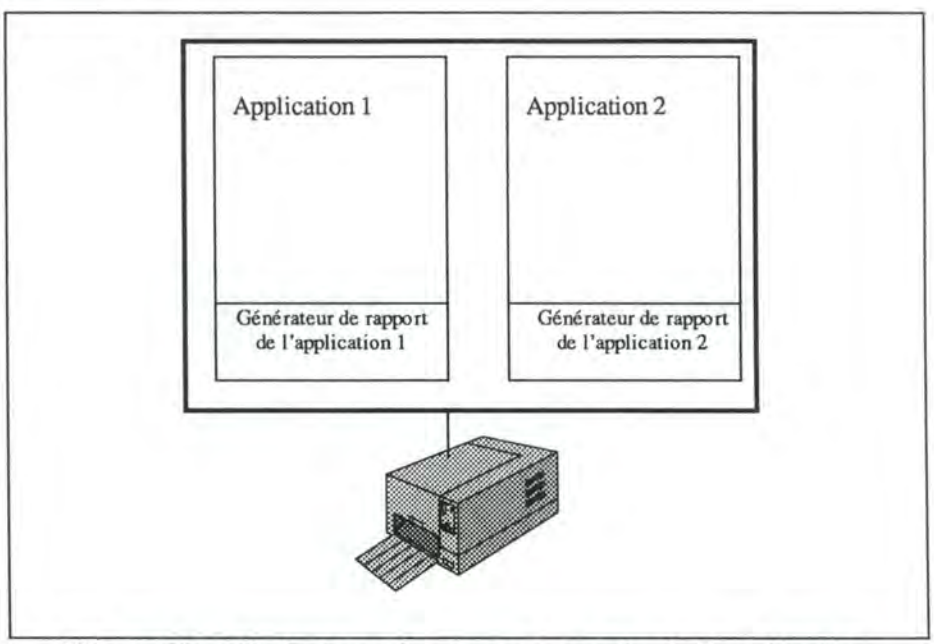

Figure 32 - Générateur de rapport au sein de l'application

Cette méthode présente elle aussi plusieurs inconvénients. D'abord, chaque application contiendra un générateur de code PostScript, qui sera probablement différent de celui des autres applications. Cela provoquera une redondance de code totalement inutile. Ensuite, le programmeur devra, pour chaque nouvelle application, se replonger dans l'étude du PDL utilisé. De plus, le changement du langage de définition de page (remplacement d'une imprimante PostScript par une imprimante PCL, par exemple) entraînera la modification de toutes les applications. Ce problème vient essentiellement du non-respect du modèle de structuration d'une application vu au Chapitre 1: la frontière entre l'interface utilisateur et les traitements est inexistante.

On le voit, cette solution est loin d'être parfaite. Il faudrait donc normaliser l'impression au sein des applications.

16 ibid.

5.3. Une librairie de fonctions d'impression

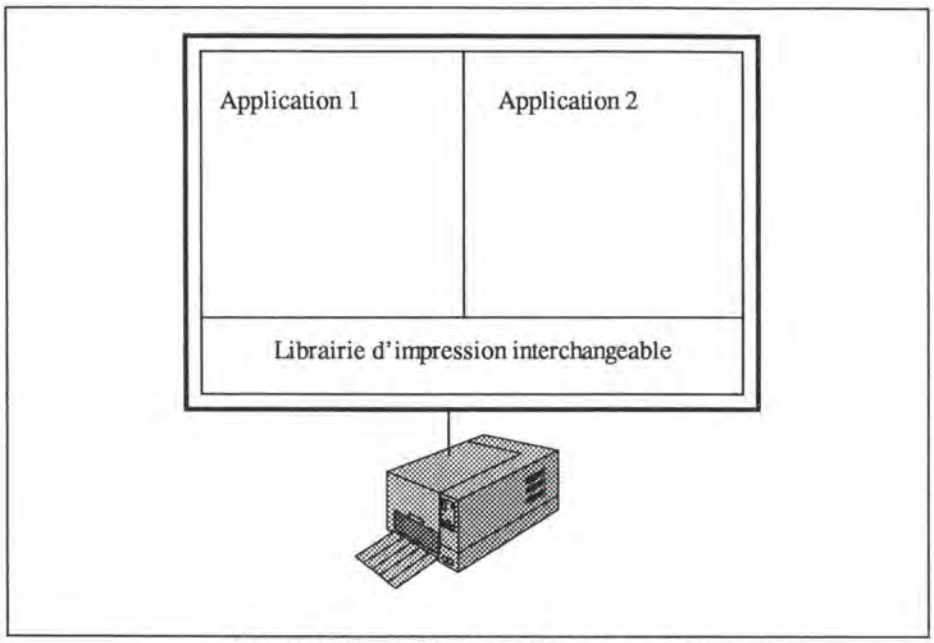

Figure 33 - Librairie d'impression

Une librairie peut reprendre un ensemble de fonctions (**DébuterDocument()**, **ImprimerTexte()**, **ImprimerCercle()**, **ImprimerRectangle()**, **AvancerPage()**, etc) permettant à chaque application l'utilisant d'accéder à l'imprimante. Ces fonctions sont sémantiquement proches de celles utilisées pour accéder à l'écran.

Le programmeur a ainsi à sa disposition une interface avec l'imprimante qu'il peut réutiliser dans chacune de ses applications. Le changement de PDL se fait de manière transparente: soit on remplace la librairie par une autre, soit la librairie reconnaît plusieurs PDL.

5.4. Une librairie de fonctions d'affichage et d'impression

On peut aller plus loin en proposant une librairie unique d'affichage et d'impression qui, selon un certain paramètre, générerait une requête du protocole X ou générerait du code d'un PDL.

La société Bristol Technology Inc. propose une telle librairie, Xprinter[17]. Celle-ci offre deux types de fonctions: des fonctions identiques à celles de la librairie Xlib (elles portent les mêmes noms à l'exception que le préfixe *X* est remplacé par *Xp*; par exemple, **XpDrawRectangle** ou **XpFillPolygon**) et des

[17] BRISTOL TECHNOLOGY INC., *Xprinter - The PostScript and PCL printer language library for the X Window System*, Bristol Technology, 1993.

fonctions spécifiques à l'impression (par exemple, **XpGetPrinterInfo** ou **XpSetResolution**).

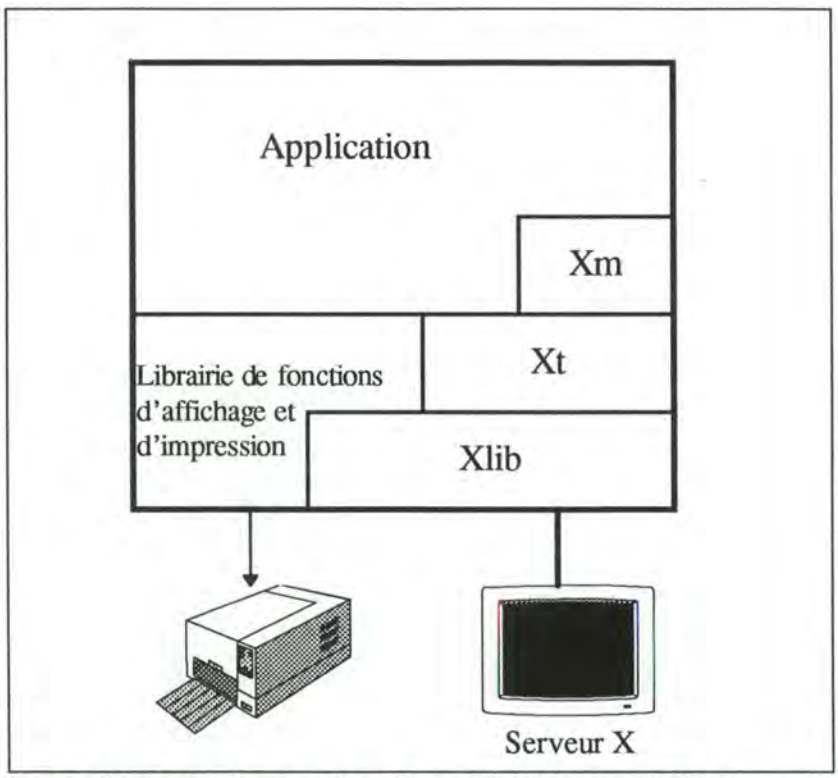

Figure 34 - Librairie de fonctions d'affichage et d'impression

Les fonctions du premier type peuvent être utilisées indifféremment en direction de l'écran ou de l'imprimante. La distinction entre les deux est faite par l'intermédiaire d'un paramètre.

La librairie Xprinter supporte PostScript Level 1 et Level 2 ainsi que PCL Level 4 et Level 5.

Séduisante au premier abord, cette solution présente toutefois des inconvénients[18]. Pour utiliser la même méthode pour l'affichage que pour l'impression, la plupart des appels à la librairie Xlib doivent se faire par l'intermédiaire de la librairie Xprinter, ce qui réduit fortement la portabilité de l'application puisque l'interface de cette librairie n'est pas standardisée. L'application ne pourra donc être exécutée que sur les plateformes pour lesquelles il existe une librairie Xprinter. De plus, le choix d'un PDL dépend des développements de Bristol Technology Inc. uniquement. Enfin, la librairie doit être liée à l'application cliente, ce qui implique un supplément de code pour chaque application utilisant la librairie.

[18] DELPERDANGE Th., DEMANGE A., GUILLAUME C., MAILLET E., MOUSEL P., POLEUR M., RETTER P., *Design and Implementation of an X Print Server*, article soumis au comité de lecture de The X Resource, p. 6.

5.5. *Une librairie d'impression d'écran*[19]

Cette solution rassemble les concepts de la copie d'écran et de la librairie d'interface. Il s'agit en effet de remplacer la librairie Xlib d'accès à un serveur X par une librairie de génération de code PDL qui contient exactement les mêmes fonctions que la Xlib. Cette librairie n'accéderait pas du tout au serveur d'affichage, mais se contenterait de transcrire les appels de l'application en code PDL. Le code de l'application cliente est inchangé: il suffit de remplacer une librairie lors du lien.

Le résultat de cette solution est qu'une application cliente affichant à l'origine des boîtes de dialogue à l'écran, le fera dorénavant sur l'imprimante. Cette solution, tout comme la copie d'écran, imprimera les barres d'ascenseur et les menus, mais les problèmes de résolution graphique s'estompent puisque le tracé est pris en charge par le PDL. Toutefois, un nouveau problème survient puisque l'utilisateur n'a plus le contrôle de l'application. En effet, à un moment ou à un autre, l'application risque d'attendre indéfiniment un événement de la part de l'utilisateur, celui-ci ne pouvant répondre puisque plus rien n'est affiché à l'écran.

19 RAVES W., *A PostScript X Server*, op. cit., p. 35

CHAPITRE 6. LA SOLUTION RETENUE: UN SERVEUR X D'IMPRESSION

6.1. Démarche

Dans toutes les solutions présentées dans le chapitre cinq, notons que l'impression d'un document est prise en charge par l'application cliente elle-même. On retrouve donc au sein de l'application, d'une part, des appels aux primitives graphiques offertes par la bibliothèque de X (Xlib) pour générer du graphisme dans une fenêtre à l'écran, et d'autre part, du code permettant de générer un graphisme identique à l'aide d'un langage de description de page, le langage PostScript par exemple, compréhensible par les périphériques d'impression.

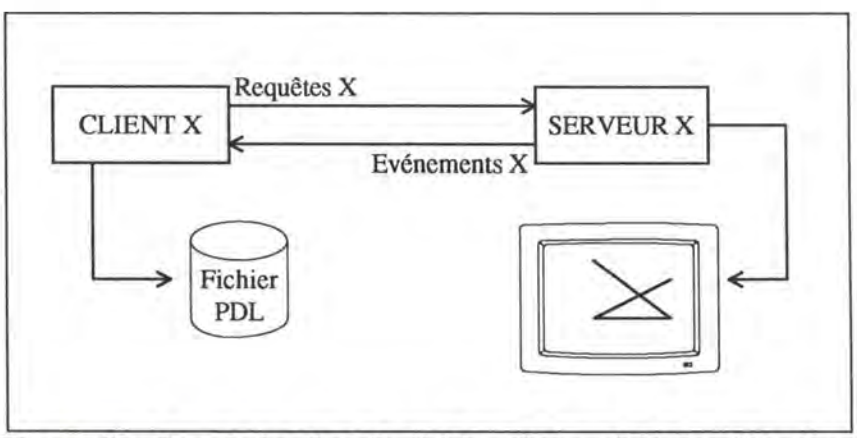

Figure 35 - La génération du fichier PDL se fait du côté client

Mais il est peut être intéressant de savoir pourquoi chaque application doit inclure une librairie de génération de code, alors qu'une seule peut s'occuper exclusivement de cette tâche. D'ailleurs, il existe déjà une application qui s'occupe de répondre aux sollicitations d'autres applications: c'est le serveur X d'affichage.

Est alors né le concept de serveur X d'impression. Une imprimante peut être vue comme un dispositif d'affichage. Dès lors, une analogie entre l'écran d'un serveur X d'affichage et une imprimante est possible. Une application voulant accéder à une imprimante, se connecte au serveur de cette imprimante et dialogue avec lui de la même manière qu'elle dialogue avec un serveur X traditionnel. Pour que cette analogie soit complète, il faut que l'API d'accès au serveur d'impression soit identique à celle du serveur d'affichage.

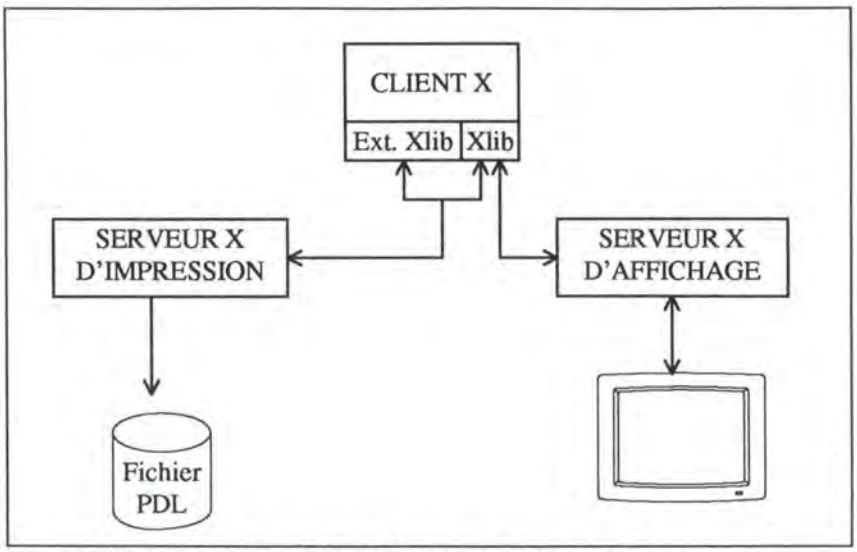

Figure 36 - Le client dialogue avec deux serveurs

Les avantages d'une telle architecture sont nombreux. L'application cliente est déchargée de l'impression ce qui entraîne un gain de temps. Par une API normalisée et transparente, l'application cliente est indépendante du langage utilisé par cette imprimante et peut donc accéder à n'importe quelle imprimante du réseau pourvu que celle-ci soit gérée par un serveur d'impression. L'application peut adresser l'écran de la même manière que l'imprimante, permettant ainsi une unification partielle du code de l'application (un dessin affiché par une séquence *s* de requêtes exigera la même séquence *s* pour l'impression).

La réalisation de ce serveur X d'impression a été menée en deux étapes. La première avait pour but de greffer sur un serveur X d'affichage un service d'impression sous forme d'extension. Pour ce faire, une nouvelle ressource a été ajoutée à la classe des *drawables* du serveur X d'affichage: la ressource *MPDocument* (Multiple Page Document). Ce nouveau drawable, qui peut être soumis à toutes les requêtes graphiques de X, se différencie des deux autres (la fenêtre et la pixmap) par sa façon de réagir aux requêtes. En effet, pour chaque requête graphique qui lui est destinée, il produit, en langage de description de page, le code correspondant.

Pour pouvoir gérer cette nouvelle ressource, trois nouvelles fonctions ont été ajoutées à la bibliothèque Xlib: une fonction permettant la création d'un MPDocument, une autre permettant de passer à la page suivante au sein d'un MPDocument, et enfin, la troisième qui désallouera et/ou imprimera un MPDocument. Naturellement, à chacune de ces nouvelles fonctions correspond une nouvelle requête du protocole X. Ce protocole a donc dû lui aussi être étendu et s'est vu ajouter trois nouvelles requêtes.

La dernière partie de cette première étape consistait à implémenter les fonctions appelées par le serveur à chaque fois que celui-ci réceptionne une

requête graphique destinée à un MPDocument. Ces fonctions ont pour but de générer dans un fichier un graphisme équivalent à l'aide d'un langage de description de page, PostScript Level 2 dans notre cas.

La deuxième étape était entièrement consacrée à la réduction du serveur X étendu pour que celui-ci ne dispose que des seules fonctionnalités d'impression. Pour y arriver, une épuration du code s'occupant de la gestion des fenêtres, des écrans, et des périphériques d'entrées s'est avérée nécessaire. On a ainsi obtenu un serveur X d'impression à part entière pouvant bien sûr fonctionner sur des machines dépourvues de périphériques d'affichage.

L'extension de la Xlib et du protocole X a été réalisé par E. Maillet[20], stagiaire au CRP-CU. Nous avons ensuite pris en charge la génération du code de description de page et la réalisation du serveur X d'impression à part entière. A ce stade, le serveur X d'impression n'est pas parfait puisqu'il ne supporte pas la couleur, la gestion des pixmaps, etc. A. Demange et C. Guillaume s'occupent de ces problèmes[21].

Avant de détailler la procédure que suit toute application cliente désireuse d'imprimer en utilisant les services d'un serveur d'impression, intéressons-nous aux caractéristiques d'utilisation et aux architectures qui sont possibles avec de tels serveurs.

Tout d'abord, à chaque imprimante connectée sur le réseau est associé un serveur d'impression qui la pilote. Ainsi, toutes les imprimantes du réseau sont accessibles par n'importe quelles applications clientes du moment que celles-ci soient autorisées à se connecter aux serveurs correspondants. Ensuite, contrairement aux serveurs d'affichage, les serveurs d'impression peuvent s'exécuter localement ou à distance. Dans le premier cas, l'imprimante est directement connectée sur la machine abritant le serveur la pilotant. Dans le second, le serveur d'impression s'exécute sur une machine et contrôle un périphérique d'impression connecté sur une autre machine. Enfin, l'utilisation des serveurs X d'impression n'exclut pas la situation dans laquelle zéro ou un serveur d'affichage côtoie sur la même machine zéro, un ou plusieurs serveurs d'impression. Celle signifie entre autre que l'on peut avoir une machine démunie d'écran et donc dépourvue de serveur d'affichage mais abritant plusieurs serveurs d'impression.

La Figure 37 résume les configurations qui sont possibles avec un serveur X d'impression. Tout d'abord, on a le cas de la machine A qui dispose de deux périphériques, un d'affichage et un d'impression. Le premier est géré par un serveur X d'affichage qui est nécessairement local, tandis que le second est piloté à distance par un serveur X d'impression. Ensuite, on trouve la machine B à laquelle sont connectés un écran et une imprimante. Ces derniers sont pilotés respectivement par un serveur X d'affichage et un serveur X d'impression qui

20 MAILLET E., *Conception d'un serveur d'impression sous X-Window*, CRP-CU, Luxembourg, 1993

21 DEMANGE A., GUILLAUME C., *Rapport de stage*, CRP-CU/MIAGE NANCY II, 1994

sont tous deux exécutés localement. Cette machine abrite encore un serveur d'impression qui est associé à une imprimante éloignée. Enfin, la machine C, est une machine dépourvue d'écran qui exécute deux serveurs X d'impression pilotant chacun deux périphériques locaux.

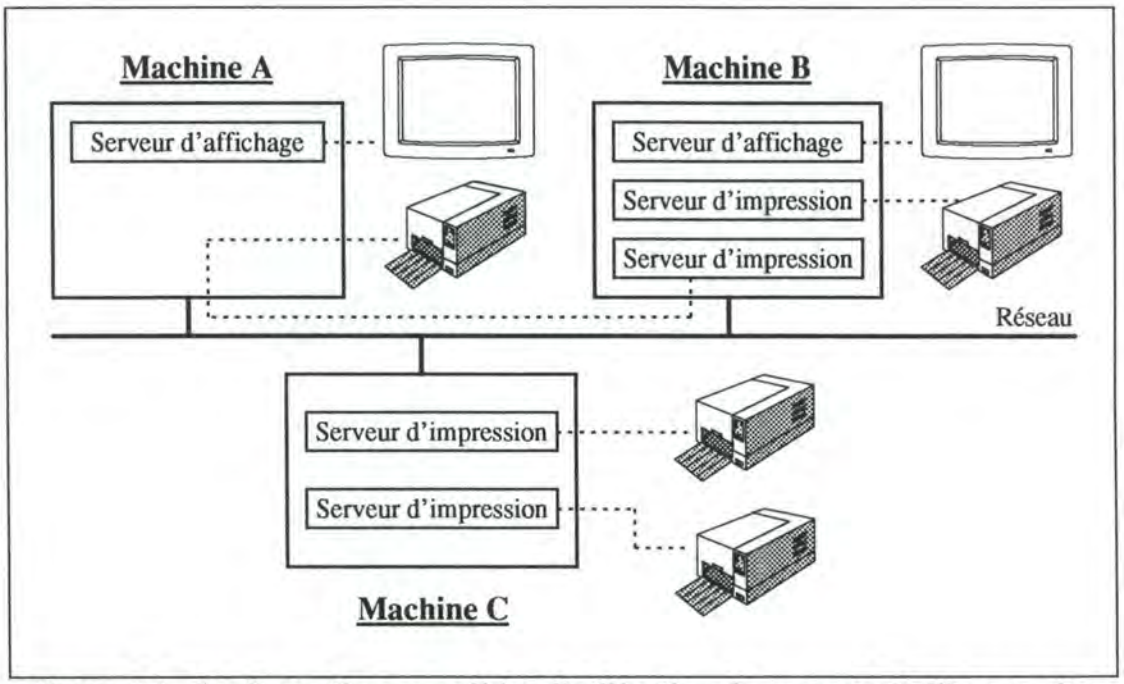

Figure 37 - Configurations possibles d'utilisation de serveurs X d'impression

Revenons maintenant sur la procédure que toute application cliente doit respecter si elle désire imprimer avec un serveur X d'impression. Avant toute chose, cette application doit connaître quelles sont les imprimantes qui lui sont directement accessibles sur le réseau. Pour s'acquitter de cette tâche, elle fait appel à une fonction particulière de la Xlib étendue qui lui renseigne les adresses des serveurs X d'impression auxquels elle est autorisée à se connecter. Ensuite, en utilisant une de ces adresses, l'application va demander une connexion au serveur d'impression correspondant. Une fois la connexion établie, l'application cliente peut envoyer toutes les requêtes graphiques destinées à un MPDocument au serveur d'impression qui se chargera de générer dans un fichier le code PostScript correspondant. La fin du document imprimé est spécifié par le client par l'envoi d'une requête particulière au serveur d'impression. Dans ce cas, ce dernier n'a plus qu'à fermer et à envoyer le fichier contenant le code PostScript à l'imprimante qu'il pilote. Pour que cela soit possible, le serveur d'impression doit connaître le nom de l'imprimante qu'il contrôle. Cette information lui est donnée par le biais de paramètres de lancement. En effet, le lancement d'un serveur X d'impression accepte deux nouveaux paramètres. Le premier (`-log`) renseigne le serveur sur le nom de l'imprimante tel qu'il est connu par le système d'exploitation. Le second (`-phy`) identifie un fichier d'extension PostScript Printer Definition (PPD) contenant des informations sur les caractéristiques physiques du périphérique d'impression. Le lancement du serveur s'effectue de la manière suivante:

```
XPsServer :display -log nom_imprimante -phy fichier_PPD
```

6.2. *Un langage de description de page: PostScript*

Avant d'entrer dans les détails de l'implémentation du serveur X d'impression, nous allons brièvement présenter le langage de description de page que nous avons utilisé, à savoir, PostScript, de la société Adobe Systems Inc.

PostScript est un langage qui permet de communiquer une description d'un document imprimable à un périphérique d'impression. Tout périphérique compatible PostScript intègre un interpréteur qui traduit les instructions PostScript en instructions compréhensibles par ce périphérique. Ce langage permet de définir des pages comprenant aussi bien du texte que du graphisme. D'ailleurs, les lettres sont gérées comme des graphiques, ce qui permet, entre autres, d'allonger, de courber, d'ombrer et de pivoter les lettres.

Un des grands avantages de PostScript est son indépendance vis-à-vis du matériel. Ainsi, le programmeur se consacre uniquement à l'effet désiré, sans se préoccuper du périphérique de destination. De plus, il est possible d'imprimer un document brouillon sur une imprimante, puis, sans changer une seule ligne de code, d'utiliser une photocomposeuse pour obtenir un document de haute qualité.

PostScript est dynamique en ce sens que de nouveaux opérateurs peuvent être définis. Les opérateurs existants peuvent être combinés pour former d'autres opérateurs. Il est ainsi possible de construire une palette de fonctionnalités de très haut niveau (par exemple, une seule commande peut être utilisée pour préparer un formulaire).

PostScript en est maintenant à sa deuxième version (Level 2). Il possède un ensemble de 400 opérateurs de base, appelées *primitives* car ce sont des tâches que l'interpréteur sait exécuter. Ces primitives sont les constituants de base d'un programme PostScript.

PostScript manipule des objets. Ces objets sont regroupés en plusieurs catégories: les noms littéraux, les noms et les procédures. Les noms littéraux sont les nombres, les caractères, les chaînes, etc. Les noms sont des étiquettes désignant d'autres objets et constituant des entrées dans des dictionnaires (voir ci-après). Les procédures sont des groupes d'objets devant être exécutés en séquence, et qui sont traités comme une seule unité.

PostScript travaille à l'aide de piles. Une de ces piles est la pile des opérandes. C'est au sommet de cette pile que PostScript trouve les données dont un opérateur a besoin. Ces données sont les opérandes de l'opérateur. Le résultat de l'exécution d'un opérateur est également placé sur cette pile. Une pile est donc un endroit de stockage temporaire. Remarquons que, à la différence d'autres langages, PostScript exige que les opérandes précèdent l'opérateur: l'interpréteur utilise la notation postfixée.

PostScript utilise également des dictionnaires. Un dictionnaire est une table qui associe un nom ou une *clé* à une valeur. C'est grâce à ce mécanisme

qu'il est possible de donner un nom à une variable ou à un nouvel opérateur. Tout comme pour les opérandes et les résultats, il existe une pile de dictionnaires.

Lorsque PostScript exécute un programme et qu'il rencontre un nom littéral (précédé du symbole spécial '/', sauf s'il s'agit de nombres), il le dépose sur la pile des opérandes. S'il rencontre un nom, il effectue une recherche dans la pile des dictionnaires, en commençant par le dictionnaire " du haut ". Si la clé ne se trouve pas dans ce dictionnaire, il passe au suivant et ainsi de suite jusqu'à ce qu'il trouve la clé ou qu'il arrive dans le bas de la pile. Dans ce dernier cas, une erreur est générée. Si PostScript trouve la clé, il vérifie le type de la valeur associée à la clé. S'il s'agit d'une procédure, il l'exécute; s'il s'agit d'un littéral, il le place sur la pile des opérandes. PostScript agit de la sorte jusqu'à la fin du programme ou jusqu'à ce qu'il rencontre une erreur.

L'unité d'impression de PostScript est la page. Il s'agit d'une page logique. En fait, elle peut correspondre à une page physique, à une partie de page physique ou à plusieurs pages physiques. La page courante est l'espace sur lequel l'interpréteur dessine. Le chemin courant est le tracé invisible d'un contour d'une lettre ou d'une forme. C'est un opérande implicite, c'est-à-dire qu'il ne faut pas l'empiler, il est automatiquement disponible. Le point courant est le dernier point du chemin courant. Des opérateurs comme `fill` (remplir une zone sur la page), `stroke` (tracer une ligne sur la page) et `show` (tracer des caractères sur la page) utilisent le chemin courant pour déterminer l'emplacement où la peinture doit être déposée. Les principaux opérateurs de construction de chemins (et qui ne tracent donc aucune marque sur la page) sont `newpath` (commencer un nouveau chemin), `moveto` (déplacer le point courant au point indiqué), `lineto` (tracer une ligne du point courant au point indiqué) et `rlineto` (tracer une ligne à partir du point courant vers un point indiqué en coordonnées relatives).

Chaque point d'une page PostScript est décrit au moyen de deux coordonnées, le couple (x,y), qui ne correspondent pas forcément aux coordonnées sur la page physique puisque la page PostScript est une page virtuelle. Ce système de coordonnées est appelé *espace utilisateur*. L'origine de cet espace, le point (0,0), se trouve dans l'angle inférieur gauche. En ce qui concerne l'orientation des axes, le sens positif sur l'axe des x correspond à un déplacement horizontal vers la droite et le sens positif sur l'axe des y correspond à un déplacement vertical vers le haut. Par défaut, une unité sur les axes est de 1/72 de pouce[22].

Voici un exemple de code PostScript:

```
/Times-Roman 12 selectfont
100 150 moveto
(Exemple de texte) show
showpage
```

[22] Il s'agit du *point*, unité de mesure des imprimeurs.

A la première ligne de cet exemple, PostScript empile successivement les valeurs Times-Roman et 12 sur la pile des opérandes. Ensuite, il exécute la procédure `selectfont` qui se charge de préparer la police courante (Times Roman 12 points dans notre cas). Ensuite, le point courant est placé aux coordonnées (100, 150). Puis, le texte " Exemple de texte " est tracé. Enfin, la commande `showpage` active l'impression de la page, puis efface la page pour en faire la nouvelle page courante vierge.

L'exemple suivant montre le tracé d'un rectangle:

```
100 100 moveto
100 200 lineto
300 200 lineto
300 100 lineto
100 100 lineto
stroke
showpage
```

La première ligne positionne le point courant. Les quatre lignes suivantes tracent le chemin courant. La commande `stroke` trace le chemin courant et la commande `showpage` imprime la page.

La définition d'un nouvel opérateur est décrite par l'exemple qui suit:

```
/TraceTexte {moveto show} def
(Coucou) 100 150 TraceTexte
(Ceci est une petite phrase.) 100 300 TraceTexte
```

La première ligne est interprétée comme suit: PostScript détecte un nom littéral, TraceTexte, qu'il place au sommet de la pile des opérandes. Il place également la procédure {moveto show} sur la pile des opérandes et il exécute la procédure def. Celle-ci utilise les deux éléments du sommet de la pile pour insérer une nouvelle entrée dans le dictionnaire courant de la pile des dictionnaires. TraceTexte devient donc une clé, correspondant à une procédure. Ensuite, le texte "Coucou" est empilé, ainsi que les valeurs 100 et 150. PostScript recherche alors la clé TraceTexte dans le dictionnaire courant et exécute la procédure associée. moveto dépile les valeurs 100 et 150 pour positionner le point courant et l'opérateur show dépile "Coucou" qu'il trace sur la page. La dernière ligne est interprètée de la même manière.

6.3. Hiérarchie du code source d'un serveur X

Pour comprendre la suite, il importe d'avoir en tête la hiérarchie des répertoires du code source d'un serveur X. Elle se présente comme suit:

X/mit/server: contient le fichier exécutable du serveur X ainsi que les répertoires spécifiques au serveur.
X/mit/server/ddx: contient le code du serveur dépendant du matériel (nous y trouvons notamment le répertoire *sun*)
X/mit/server/dix: contient le code du serveur indépendant du matériel (nous y trouvons la fonction principale **main()** du serveur)
X/mit/server/os: contient le code spécifique au système d'exploitation
X/mit/extensions: contient les répertoires des différentes extensions du serveur.

La réalisation d'un serveur X d'impression a impliqué deux modifications dans cette hiérarchie: l'ajout d'une extension au serveur, dans *X/mit/extensions/xmpdoc*, et l'ajout d'une partie dépendante du matériel, dans *X/mit/server/ddx/printer*. Soulignons que l'extension du côté serveur est indépendante du langage de description de page généré; c'est donc dans la partie dépendante du matériel que sera généré le code PostScript (ou tout autre langage). Il est également important de noter que nos développements s'intègrent parfaitement dans la hiérarchie de la distribution du MIT (sous forme des deux nouveaux répertoires) et qu'aucun autre fichier n'a été modifié (excepté les fichiers de compilation et un fichier dans lequel s'effectuent les initialisations des extensions du serveur).

6.4. *Extension du serveur*

Dans un premier temps, nous avons utilisé le principe des extensions offert par le protocole X (voir § 2.8) pour pouvoir créer et gérer un nouveau type de ressource: le *MPDocument* (MultiPage Document) qui est un drawable tout comme la *pixmap* et la *fenêtre*. Le serveur, le protocole X et la librairie client Xlib ont donc été étendus. Une application de test a également été développée.

Extension de la librairie Xlib et du protocole X

A la différence de la pixmap et de la fenêtre, un document imprimable contient plusieurs parties: les pages. Il manque donc au protocole standard quelques requêtes pour gérer cette caractéristique: une requête de création d'un MPDocument, une requête qui permet de passer à la page suivante et une requête qui termine le document. En conséquence, le protocole X a été augmenté de ces trois nouvelles requêtes dont les nouvelles fonctions apparentées sont:

- une fonction de création d'un MPDocument

MPDocument mpid = XMPDocumentCreate(Display display, unsigned int width, unsigned int length)

display: identificateur du serveur X
width: largeur des pages du MPDocument
length: longueur des pages du MPDocument
mpid: identificateur du nouveau MPDocument

Cette fonction accepte en entrée l'identificateur du serveur auquel l'application est connectée (connexion établie par l'appel de la fonction idoine de la librairie Xlib) ainsi que la largeur et la hauteur de la page (exprimées en 1/72 de pouce). En sortie, la fonction retourne l'identificateur du nouveau document. Dès lors, l'application peut appeler les fonctions graphiques de la Xlib en passant cet identificateur en paramètre. La Figure 38 montre le code de cette fonction qui garnit d'abord une structure représentant la requête. Ensuite, elle demande au serveur X un nouvel identificateur (par l'appel d'une fonction de la Xlib, **XAllocID()**). La requête est alors prête à être envoyée.

```
MPDocument XMPDocumentCreate(dpy, width, length)
     register Display *dpy;
     unsigned int width, length;
{
     XExtDisplayInfo *info=find_display(dpy);
     MPDocument mpid;
     register xMPDocumentCreateReq *req;

     MPDocumentSimpleCheckExtension(dpy, info, 0);

     LockDisplay(dpy);
     GetReq(MPDocumentCreate, req);
     req->reqType = info->codes->major_opcode;
     req->mpdocumentReqType = X_MPDocumentCreate;
     req->mpwidth = width;
     req->mplength = length;
     mpid = req->mpid = XAllocID(dpy);
     UnlockDisplay(dpy);
     SyncHandle();
     return(mpid);
}
```

Figure 38 - Code de la fonction d'envoi de la requête de création d'un MPDocument

• une fonction de saut de page dans un MPDocument

XMPDocumentShowpage(Display display, MPDocument mpid)

display: identificateur du serveur d'impression
mpid: identificateur du MPDocument

L'appel de cette fonction provoque un saut de page dans le document dont l'identificateur est passé en paramètre, en plus de l'identificateur du serveur auquel l'application est connectée.

• une fonction pour terminer un MPDocument, c'est-à-dire le supprimer ou l'imprimer et le supprimer

XMPDocumentFree(Display display, MPDocument mpid, Bool print)

display: identificateur du serveur d'impression
mpid: identificateur du MPDocument
print: drapeau indiquant si le document doit ou non être imprimé avant sa suppression

Si le paramètre *print* contient la valeur 1, cette fonction provoque l'impression du document identifié par *mpid* et géré par le serveur X identifié par *display*. Sinon, le document n'est pas imprimé. Dans les deux cas, après appel de la fonction, le document n'est plus accessible et l'identificateur *mpid* n'a plus de signification.

Ces fonctions sont rassemblées dans le fichier *XMPDocument.c* du répertoire *X/mit/extensions/xmpdoc/lib*, et compilées dans la librairie *libXMPDocument.a* du même répertoire. Cette librairie peut être liée à toute application cliente désirant faire appel à un service d'impression.

Le fichier *mpdocumentstr.h* contient les structures des trois nouvelles requêtes. Par exemple, voici la structure de la requête générée par la fonction **XMPDocumentCreate()**:

```
typedef struct _MPDocumentCreate {
   CARD8     reqType;
   CARD8     mpdocumentReqType;
   CARD16    length B16;
   CARD32    mpid B32;
   CARD16    mpwidth B16;
   CARD16    mplength B16;
} xMPDocumentCreateReq;
```

Le champ *reqType* désigne le code opératoire majeur (ou *major opcode*) de notre extension. Le champ *mpdocumentReqType* est destiné à recevoir le code opératoire mineur (ou *minor opcode*) permettant de distinguer la fonction appelée de notre extension. Ce code est fixé par des `#define` au début du fichier *mpdocumentstr.h*: `X_MPDocumentCreate`, `X_MPDocumentFree` et `X_MPDocumentShowpage` sont ainsi définis. Le champ *length* contient la longueur de la requête, celle-ci pouvant bien sûr varier d'un type de requête à l'autre. Le champ *mpid* identifie le document concerné par cette requête. Les deux derniers champs, *mpwidth* et *mplength*, sont les deux paramètres désignant respectivement la largeur et la hauteur de la page du MPDocument.

Extension du côté serveur

Du côté du serveur, nous l'avons vu, nous avons ajouté une extension. Pour cela, la fonction **InitExtensions**() (appelée par la fonction **main**() du serveur) du fichier *X/mit/server/ddx/mi/initext.c* s'est vue ajouter quelques lignes:

```
#ifdef MPDOC
   MPDocumentExtensionInit();
#endif
```

Cette fonction **MPDocumentExtensionInit()**, définie dans le fichier *MPDocument.c* du répertoire *X/mit/extensions/xmpdoc/server*, effectue plusieurs actions:

- Tout d'abord, elle crée un nouveau type de ressource (MPDocumentType) par l'appel de la fonction **CreateNewResourceType().** Cette fonction reçoit en entrée l'adresse d'une de nos fonctions (**MPDocFree()**), qui sera appelée lorsqu'une ressource de type MPDocumentType sera libérée. A partir de ce moment, le type MPDocument existe au même titre que la fenêtre ou que la pixmap.

- Ensuite, elle signale au restant du serveur qu'une nouvelle extension est disponible. Cela se fait par l'appel de la fonction **AddExtension().** Celle-ci reçoit en entrée le nom de l'extension (une chaîne de caractères valant "MPDOCUMENT"), le nombre d'évènements susceptibles d'être générés par l'extension (zéro dans notre cas), le nombre d'erreurs susceptibles d'être générées par l'extension (zéro dans notre cas puisque les erreurs éventuellement générées sont communes à d'autres parties du serveur; il n'y a donc pas de nouveaux types d'erreurs), l'adresse de la fonction à placer dans une nouvelle entrée du vecteur ProcVector (voir § 2.8; dans notre cas, **ProcMPDocDispatch()**), l'adresse de la fonction similaire à la précédente mais gérant l'inversion de bytes, l'adresse de la fonction appelée lorsque le serveur se réinitialise et l'adresse de la fonction **StandardMinorOpcode()** qui retourne le code opératoire mineure contenu dans une requête.

- Enfin, elle appelle la fonction **MPDocModifyProcVector()** qui modifie trois entrées dans le vecteur ProcVector. Ces entrées correspondent aux fonctions de création d'un contexte graphique (GC), de copie d'une zone et de copie d'un seul plan d'une zone. Nous l'avons vu (§2.4), un GC contient les adresses des fonctions graphiques. Le MPDocument étant un drawable particulier, ces fonctions doivent être réécrites. Ce sont ces fonctions qui généront le code PDL. En conséquence, la fonction de création d'un GC doit assigner au nouveau GC l'adresse de nos fonctions. Ces fonctions sont **MPDocCopyArea(), MPDocCopyPlane(), MPDocPolyPoint(), MPDocPolyLine(), MPDocPolySegment(), MPDocPolyRectangle(), MPDocPolyArc(), MPDocFillPoly(), MPDocPolyFillRect(), MPDocPolyFillArc(), MPDocPolyText8(), MPDocPolyText16(), MPDocImageText8(), MPDocImageText16()**. Nous verrons leur contenu plus loin.

Lorsque le serveur réceptionne une requête provenant d'un client, il repère le code opératoire majeur et l'utilise comme index dans le vecteur ProcVector. S'il s'agit d'une requête adressée à notre extension, c'est donc notre fonction **ProcMPDocDispatch()** qui est invoquée. Cette fonction, selon la valeur du code opératoire mineur, appelle à son tour une des trois fonctions suivantes: **ProcMPDocumentCreate(), ProcMPDocumentFree()** ou

ProcMPDocumentShowpage(). Voyons plus en détail le contenu de ces fonctions:

- **ProcMPDocumentCreate().** Cette fonction correspond, du côté de l'application cliente, à l'appel de la fonction **XMPDocumentCreate().** Tout d'abord, elle alloue une structure (de type MPDocumentRec) qui permettra à notre extension de gérer un MPDocument. Cette structure contient les informations suivantes: une structure DrawableRec standard où le serveur pourra trouver des informations concernant la ressource, l'identificateur du fichier prologue, l'identificateur du fichier corps (voir ci-après la signification de ces fichiers), le nom du premier fichier (le second portant le même nom suivi de la lettre 'p'; ce nom, unique, est généré grâce à la fonction C **tempnam()**), un compteur de référence (inutilisé dans cette première version), un drapeau indiquant si l'on se trouve au début d'une page, un compteur de pages, un pointeur conservant le nom de la police de caractères courante et un pointeur vers une liste des noms de polices utilisées dans le document. Après avoir initialisé cette structure, notre fonction crée les deux fichiers: le fichier corps destiné à contenir la description des pages exprimé dans un PDL et le fichier prologue destiné à contenir les initialisations préalables, également en PDL. Ces deux fichiers seront concaténés avant d'être envoyés à l'imprimante. Ensuite, une fonction d'initialisation est appelée (**MPDocInitPDLFile()**). Cette fonction peut, par exemple, écrire certaines initialisations propres au PDL. Nous verrons plus loin de quoi il s'agit. Enfin, notre fonction appelle la fonction **AddResource()** qui officialise la nouvelle ressource MPDocument.

- **ProcMPDocumentFree()**. Cette fonction correspond, du côté de l'application cliente, à l'appel de la fonction **XMPDocumentFree()**. D'abord, elle libère l'espace utilisé par la liste des noms des polices de caractères utilisées dans le document. Ensuite, elle signale au serveur que la ressource doit être détruite, par l'appel de la fonction **FreeResource()**. Cette dernière se charge d'appeler la fonction **MPDocFree()** dont nous avions donné l'adresse lors de l'appel de **CreateNewResourceType()**. **MPDocFree()** concatène le fichier corps au fichier prologue et libère l'espace utilisé par la structure MPDocumentRec. **ProcMPDocumentFree()** vérifie ensuite que l'application cliente désire imprimer le document (paramètre *print* lors de l'appel de **XMPDocumentFree()**). Dans ce cas, la fonction demande au système d'exploitation d'exécuter le fichier de commandes *XPsScript* avec, comme paramètres, le nom de l'imprimante (spécifié par le paramètre *-log* lors du lancement du serveur) et le nom du fichier. Habituellement, ce fichier de commandes contiendra donc des instructions d'impression, spécifiques au système d'exploitation utilisé. Dans le cadre de nos tests, nous avons utilisé ce fichier de commandes pour exécuter un logiciel de visualisation de fichier PostScript.

- **ProcMPDocumentShowpage()**. Cette fonction, correspondant à la fonction **XMPDocumentShowpage()**, se contente d'appeler une fonction de génération de code PDL, **MPDocWriteShowpage()**.

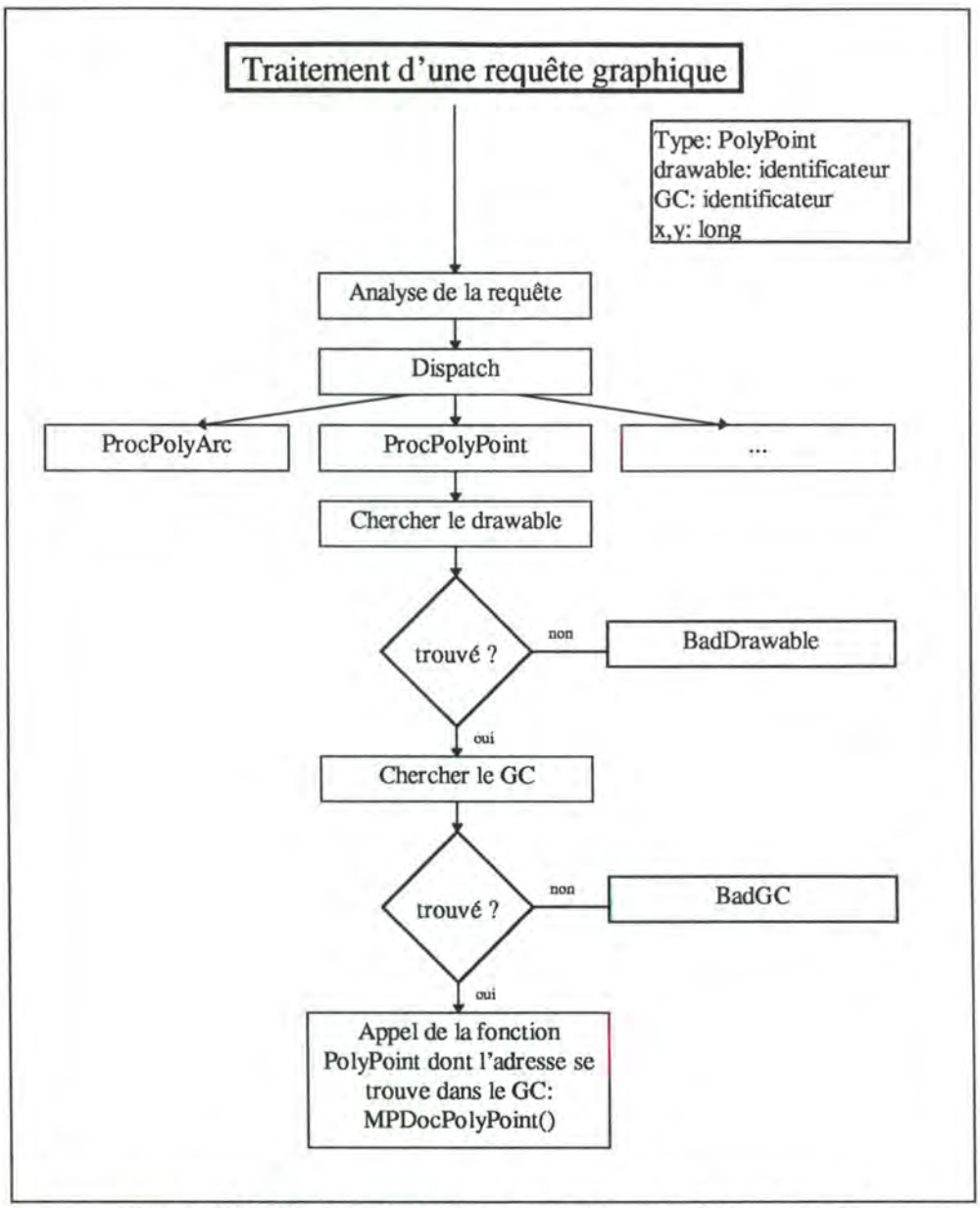

Figure 39 - Traitement d'une requête graphique par le serveur

Jusqu'à présent, nous avons vu comment se déroulait la réception d'une requête faisant partie de notre extension. Voyons maintenant la réception d'une requête graphique faisant partie du protocole X standard.

En fait, peu de choses diffèrent. Le serveur reçoit une requête et en extrait le code opératoire majeur. Il l'utilise comme index dans le vecteur ProcVector pour retirer l'adresse d'une fonction. Et il appelle cette fonction. Par exemple, s'il s'agit d'une requête de tracé de ligne, il appelle la fonction **ProcPolyLine()**. Cette fonction, après quelques tests, appelle la fonction dont l'adresse se trouve dans le GC spécifié dans la requête. C'est cette dernière fonction qui effectuera le tracé proprement dit. Ainsi, si le GC contient les adresses de nos fonctions de génération de PDL, c'est notre fonction **MPDocPolyLine()** qui sera appelée.

Les fonctions de génération de code PDL

Voyons maintenant de plus près la génération de code PDL. Nous avons vu que la partie *X/mit/extensions/xmpdoc* est indépendante du PDL utilisé. Dès lors, la génération de code doit se faire dans la partie dépendante du matériel, dans le répertoire *X/mit/server/ddx/printer/mpdoc*.

Examinons d'abord la fonction **MPDocInitPDLFile()** qui, rappelons-le, est appelée par la fonction **ProcMPDocumentCreate()**. Le code de cette fonction se trouve dans le fichier *X/mit/server/ddx/printer/mpdoc/mpdocdummy.c*.

```
void MPDocInitPDLFile(pMP)
     MPDocumentPtr pMP;
{
#ifdef MPDocInPostscript

fprintf(pMP->finit, "0 %d translate\n/print {true upath [0 0 0 0 0 0]
currentmatrix [0 0 0 0 0 0] defaultmatrix invertmatrix ustroke} def\n/paint
{true upath ufill} def\n", pMP->drawable.height);
fprintf(pMP->finit, "4 setmiterlimit\n");

#else

fprintf(pMP->finit, "MPDocPolyLine : support for non PostScript devices not
implemented\n");

#endif
}
```

MPDocInitPDLFile() écrit dans le fichier prologue des définitions PostScript utiles pour les requêtes graphiques. Ainsi, l'instruction `0 %d translate` est très importante car c'est elle qui permet, en partie, de transformer des coordonnées X en coordonnées PostScript. En effet, l'origine (0,0), en PostScript, se trouve dans le coin inférieur gauche, alors que l'origine de l'espace utilisateur X se trouve dans le coin supérieur gauche. Il faut donc agir sur l'axe des *y* uniquement. L'opérateur PostScript `0 t translate` déplace le point d'origine dans le coin supérieur gauche si *t* est la hauteur, en points, de l'espace utilisateur. Il subsiste un problème puisque l'axe des *y* est inversé. Toute coordonnée *y* devra donc être précédée du signe '-'.

Le reste du code PostScript définit de nouveaux opérateurs `paint` et `print`, utilisés pour le tracé d'arcs.

Notons que toutes les fonctions de génération de code sont structurées de telle façon qu'un nouveau PDL puisse être facilement pris en compte: un #ifdef permet au compilateur de n'inclure que certaines parties du programme. Un serveur prévu pour gérer un périphérique PostScript n'est compilé qu'avec la partie du programme générant du code PostScript.

Nous avons également vu que la fonction **ProcMPDocumentShowpage()** appelait la fonction **MPDocWriteShowpage()** dont le code se trouve dans le fichier *X/mit/server/ddx/printer/mpdoc/mpdocdummy.c*.

```
void MPDocWriteShowpage(pMPDocument)
      MPDocumentPtr pMPDocument;
{
if (pMPDocument->IsNewPage)
      fprintf(pMPDocument->fd, "gsave showpage grestore\n");
else {
      fprintf(pMPDocument->fd, "showpage grestore\n");
      pMPDocument->IsNewPage = xTrue;
     }
}
```

Cette fonction effectue un traitement particulier selon que l'on se trouve en début d'une nouvelle page ou non. Dans ce dernier cas, il faut mettre à *vrai* le champ *IsNewPage* de la structure MPDocumentRec.

Les requêtes graphiques du protocole X sont les suivantes: CopyArea (copier une zone vers une autre), CopyPlane (copier un plan vers un autre), PolyPoint (tracer des points), PolyLine (tracer des lignes adjacentes), PolySegment (tracer des segments), PolyRectangle (tracer des rectangles), PolyArc (tracer des arcs de cercles), FillPoly (remplir une figure circonscrite par des lignes adjacentes), PolyFillRect (remplir des rectangles), PolyFillArc (remplir des arcs de cercles), PolyText8 (tracer du texte codé sur 8 bits), PolyText16 (tracer du texte codé sur 16 bits), ImageText8 (tracer du texte codé sur 8 bits) et ImageText16 (tracer du texte codé sur 16 bits). Pour chacune de ces requêtes, nous avons écrit une fonction de génération de code PDL. Prenons, par exemple, la fonction **MPDocPolyLine()** qui répond à la requête PolyLine. Elle reçoit en entrée deux informations communes à toutes les requêtes graphiques, à savoir l'adresse de la structure identifiant le drawable (notre structure de type MPDocumentRec) et l'adresse du GC, ainsi que trois informations spécifiques à la requête graphique, à savoir, le mode (indiquant la façon de nommer les coordonnées: relatives ou absolues), le nombre de points à tracer et l'adresse du tableau contenant les coordonnées des points.

```
void MPDocPolyLine(pDrawable, pGC, mode, npt, pptInit)
      register DrawablePtr pDrawable;
      GCPtr pGC;
      int mode;
      int npt;
      xPoint *pptInit;
{
File *f;

if (npt < 2) return;

f = ((MPDocumentPtr)pDrawable)->fd;
```

```
#ifdef MPDocInPostScript

psInitPaint(pGC, f, (MPDocumentPtr)pDrawable);
psPolyLine(pDrawable, pGC, mode, npt, pptInit, f);
if (pGC->lineStyle == LineDoubleDash)
      psDoubleDash(pGC, f);

fprintf(f, "stroke\n");

#else

/* Implementations in other page description languages go here */
fprintf(f, "MPDocPolyLine: support for non PostScript devices not
implemented yet.n");

#endif
}
```

La fonction **MPDocPolyLine()** appelle d'abord la fonction **psInitPaint()** qui initialise une série de caractéristiques selon les valeurs contenues dans le GC. Notons que tous les champs du contexte graphique ne sont pas encore pris en compte. Il s'agit notamment des couleurs d'arrière et d'avant-plan, la pixmap de clipping et de remplissage et la fonction de tracé (ou, et, oux, etc.)

```
void psInitPaint(pGC, f, pMPDocument)
      GCPtr pGC;
      FILE *f;
      MPDocumentPtr pMPDocument;
{
int tmp,i;

if (NewPage(f, pMPDocument) == BadAlloc) return;
switch(pGC->capStyle) {
      case CapButt: tmp = 0;
            break;
      case CapRound: tmp = 1;
            break;
      case CapProjecting: tmp = 2;
            break;
      default: tmp = 0;
            break;
};
fprintf(f, "%d setlinecap\n", tmp);

switch(pGC->joinStyle) {
      case JoinMiter: tmp = 0;
            break;
      case JoinRound: tmp = 1;
            break;
      case JoinBevel: tmp = 2;
            break;
      default: tmp = 0;
            break;
};
fprintf(f, "%d setlinejoin\n", tmp);

switch(pGC->lineStyle) {
      case LineSolid: fprintf(f, "[] 0 setdash\n");
            break;
```

```
      case LineOnOffDash: fprintf(f, "[");
            for(i=0;i<pGC->numInDashList;i++)
                  fprintf(f, "%d ", (int) pGC->dash[i]);
            fprintf(f, "] %d setdash\n", pGC->dashOffset);
            break;
      case LineDoubleDash: /* Special treatment in psDoubleDash function */
            break;
      default: break;
};
fprintf(f, "%d setlinewitdh\n", pGC->lineWidth);
}
```

L'opérateur `setlinecap` du PDL PostScript permet de spécifier la manière dont les extrémités d'une ligne sont dessinées (arrondies, carrées étendues, carrées strictes). `setlinejoin`, pour sa part, spécifie la façon dont deux lignes se joignent (arrondies, tronquées, étendues). Le champ *lineStyle* du GC spécifie la manière dont est tracée la ligne (pleine ou pointillée), ce qui se fait par l'opérateur PostScript `setdash`. Enfin, `setlinewidth` spécifie la largeur du trait.

Notons que la fonction **psInitPaint()** appelle d'abord la fonction **NewPage()**. Cette fonction examine le champ *IsNewPage* du drawable MPDocument pour savoir si l'on se trouve en début de page. Dans ce cas, il faut augmenter le compteur de pages, placer la valeur *false* dans le champ *IsNewPage* et supprimer la référence à la police courante (aucune police n'a encore été sélectionnée pour cette nouvelle page).

La fonction **MPDocPolyLine()** appelle ensuite la fonction **psPolyLine()**:

```
void psPolyLine(pDrawable, pGC, mode, npt, pptInit, f)
      register DrawablePtr pDrawable;
      GCPtr pGC;
      int mode;
      int npt;
      xPoint *pptInit;
      FILE *f;
{
int i;

if (pGC->miTranslate)
      for(i=0;i<npt;i++) {
            pptInit[i].x += ((MPDocumentPtrpDrawable->drawable.x;
            pptInit[i].y += ((MPDocumentPtrpDrawable->drawable.y;
      };
if (mode == CoordModePrevious)
      for(i=0;i<npt;i++) {
            pptInit[i].x += pptInit[i-1].x;
            pptInit[i].y += pptInit[i-1].y;
      };
fprintf(f, "%d -%d moveto\n", pptInit[0].x, pptInit[0].y);
for(i=1;i<npt-1;i++)
      fprintf(f, "%d -%d lineto\n", pptInit[i].x, pptInit[i].y);

if ((pptInit[0].x == pptInit[i].x) && (pptInit[0].y == pptInit[i].y))
      fprintf(f,"closepath\n");
else fprintf(f, "%d -%d lineto\n", pptInit[i].x, pptInit[i].y);
};
```

Cette fonction construit le chemin courant. Elle vérifie d'abord si les points du tableau sont exprimés en coordonnées relatives ou absolues par rapport à l'espace utilisateur (*miTranslate*). Ensuite, elle vérifie (*mode*) si les points sont exprimés en coordonnées relatives (par rapport au précédent point) ou absolues (par rapport à l'origine de l'espace utilisateur). Ensuite, le chemin courant est effectivement construit au moyen des opérateurs PostScript `lineto` et/ou `closepath`, ce dernier permettant de prolonger le chemin courant entre le dernier point (c'est-à-dire le point courant) et le premier point de ce chemin.

Remarquez le signe '-' devant la coordonnée *y* X qui permet de la transformer en coordonnée *y* PostScript.

Les autres fonctions de génération de code sont similaires à **MPDocPolyLine()**.

Notons quand même la fonction **MPDocFillPoly()** qui appelle la fonction **psPolyLine()**. Mais, au lieu d'ajouter l'opérateur `stroke` ou `closepath` pour tracer le contour, elle écrit l'opérateur `fill` ou `eofill` pour remplir la figure formée par les points. La différence entre ces deux derniers opérateurs réside dans la façon de traiter les lignes qui se croisent.

Cette fonction **psPolyLine()** est aussi utilisée par **MPDocPolySegment()**, **MPDocPolyRectangle()** et **MPDocPolyFillRect()**. De même, une fonction **psPolyArc()** est commune aux deux fonctions **MPDocPolyArc()** et **MPDocPolyFillArc()**.

La fonction **MPDocPolyText8()** est un peu particulière. En effet, la police de caractères est une source de problèmes puisque une police X n'a pas forcément d'équivalent PostScript dans le périphérique utilisé et, si elle l'a, cette police PostScript ne porte pas le même nom. Ainsi, la police X de nom `-adobe-times-bold-r-normal--0-240-75-75-p-0-iso8859-1` a comme équivalent PostScript une police Times-Bold. Comme il n'existe aucun moyen de déterminer le nom PostScript à partir d'un nom X, nous avons utilisé un fichier (*fontmap*) établissant la correspondance entre une police X et une police PostScript.

Notre fonction **MPDocPolyText8()** recherche d'abord le nom complet de la police de caractères X identifiée par un champ du contexte graphique. A partir de ce nom, elle en génère un autre, un nom mininal qui ne contient plus que les caractéristiques suivantes: le vendeur, la famille, le poids (gras, etc), l'inclinaison (roman, italique), la taille de la police (normale, large, etc) et le style (serif, sans serif, etc). Par exemple, `-adobe-times-bold-r-normal--0-240-75-75-p-0-iso8859-1` devient `adobe-times-bold-r-normal-`. Le nom complet n'est pas nécessaire car, sous X et contrairement à PostScript, une police en taille 10 points est différente de la même police en 12 points. Il n'est donc pas utile de se préoccuper de la taille de la police. Ensuite, la fonction vérifie si on doit changer la police courante. Si c'est le cas, elle recherche dans le fichier de correspondance

la police PostScript qui doit être utilisée, avec le nom minimal comme critère de recherche.

Le fichier de correspondance admet deux types d'entrée: la police PostScript correspondant à la police X existe dans l'imprimante ou elle n'existe pas dans l'imprimante mais sa définition PostScript peut être trouvée dans un fichier. La syntaxe est donc la suivante: soit `<Police_X> = <Police_PS>`, soit `<Police_X> = [<Fichier_PS> = <Police_PS>]`. Le caractère '#' est utilisé pour indiquer le début de commentaires. Voici un exemple de fichier *fontmap*:

```
adobe-times-medium-r-normal- = Times-Roman
adobe-times-medium-i-normal- = Times-Italic     # commentaires
adobe-times-bold-r-normal- = Times-Bold
adobe-times-bold-i-normal- = Times-BoldItalic
bitstream-charter-medium-r-normal- =
[/usr/local/lib/ghostscript/fonts/bchr.gsf=CharterBT-Roman]
bitstream-charter-bold-i-normal- =
[/usr/local/lib/ghostscript/fonts/bchbi.gsf=CharterBT-BoldItalic]
```

Dans cet exemple, la police X `adobe-times-medium-r-normal-` est connue, dans l'imprimante, sous le nom de Times-Roman. Mais la police `bitstream-charter-bold-i-normal-` n'existe pas dans l'imprimante. Le fichier *bchbi.gsf* contient la définition de cette police. Une fois ce fichier chargé par l'imprimante, la police sera connue sous le nom de CharterBT-BoldItalic.

Revenons au déroulement de la fonction **MPDocPolyTexte8()**. Après avoir recherché la police dans le fichier de correspondance, elle effectue un traitement particulier selon le type de la police. Par type, nous entendons: la police est décrite dans un fichier, la police est une police native, la police n'existe pas dans le fichier de correspondance. Si la police est décrite dans un fichier, ce fichier est concaténé au fichier prologue. Si la police est native, il n'y a rien à faire. Si la police n'existe pas, on utilise la police par défaut ("Courier" en l'occurence).

Ensuite, le nom de la nouvelle police est sauvegardé dans la structure MPDocumentRec et devient la police par défaut de la page PostScript courante. Le code PostScript de changement de police est écrit dans le fichier corps, suivi du code d'impression du texte proprement dit. S'il est important de garder une trace des polices utilisées dans le document, c'est pour ne pas inclure plusieurs fois la définition d'une police de caractères.

En résumé

En guise de résumé, la Figure 40 montre la correspondance des fonctions de l'application cliente et du serveur. Les fonctions de l'application cliente **XMPDocumentCreate()**, **XMPDocumentShowpage()** et **XMPDocumentFree()** correspondent respectivement aux fonctions du serveur **ProcMPDocumentCreate()**, **ProcMPDocumentShowpage()** et **ProcMPDocumentFree()**. Les fonctions de la Xlib **XDrawLine()**, **XDrawRectangle()**, etc. correspondent à l'appel des fonctions du serveur **MPDocPolyLine()**, **MPDocRectangle()**, etc.

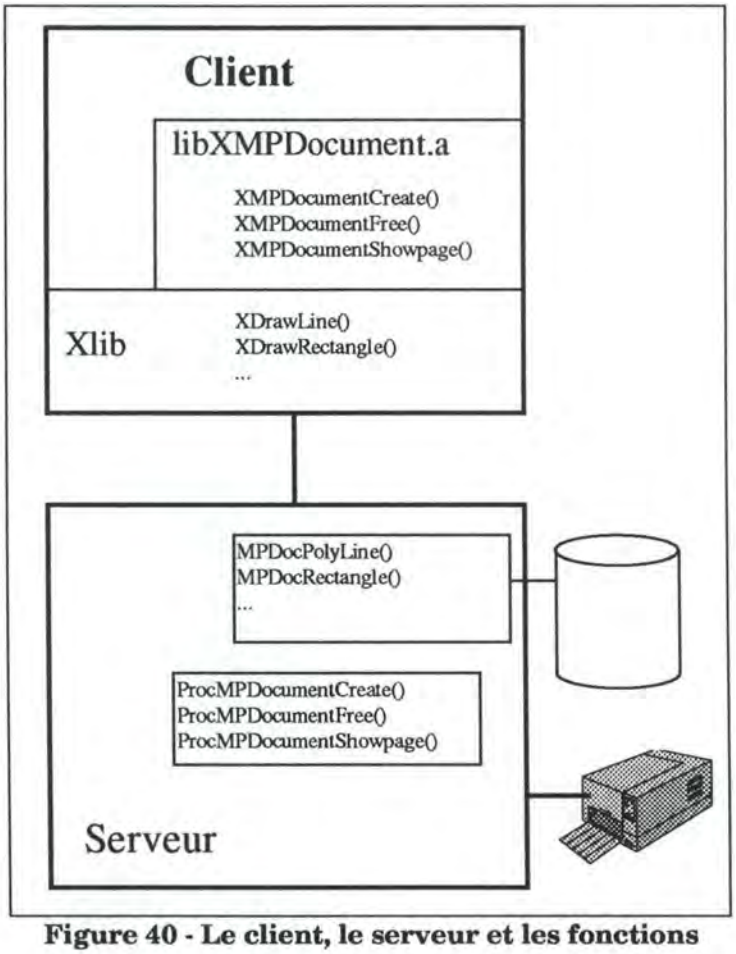

Figure 40 - Le client, le serveur et les fonctions

Une application test

Afin de tester le nouveau drawable MPDocument, nous avons développé une application cliente. Elle permet à l'utilisateur de créer un MPDocument, de passer à la page suivante et d'imprimer le document. Lorsqu'un document est créé, l'utilisateur a la possibilité de dessiner à l'écran de manière interactive. Il dispose d'un menu lui permettant de choisir la fonction graphique qu'il désire utiliser: tracer des lignes, des points, des arcs, des rectangles, des segments, des arcs remplis, des rectangles remplis, ainsi que du texte. Dès qu'il a choisi une fonction, une boîte de dialogue s'affiche dans laquelle il peut initialiser tous les

composants du contexte graphique. Ensuite, l'utilisateur utilise la souris pour tracer l'objet de son choix. Par exemple, s'il veut tracer une ligne, il clique une première fois pour désigner le premier point. A partir de ce moment, une ligne en pointillé partant du dernier point tracé et allant jusqu'au curseur permet à l'utilisateur d'avoir un aperçu de ce que serait la ligne s'il cliquait à cet endroit[23].

Trois drawables sont déclarés dans cette application: la fenêtre qui permet à l'utilisateur de dessiner, la pixmap qui permet la gestion des recrouvrements de la fenêtre et le document. Toutes les requêtes graphiques commandées par l'utilisateur sont donc envoyées aux trois drawables.

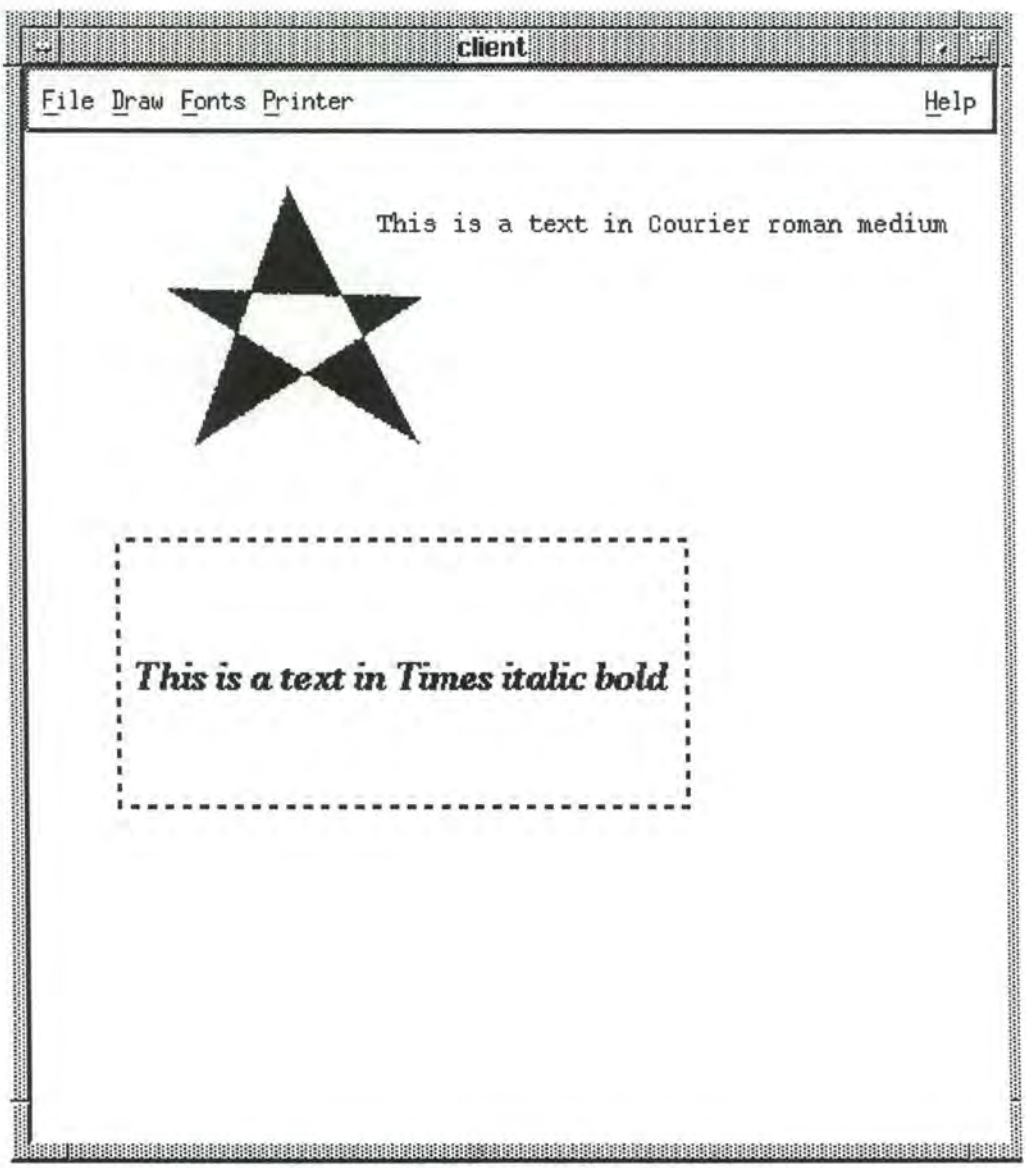

Figure 41 - L'application de test

[23] Il s'agit de la technique du *Rubber Band*.

L'impression d'un document ou sa visualisation par un utilitaire de type Ghostscript permet à l'utilisateur de s'assurer que ce qui est imprimé correspond bien à ce qu'il a dessiné.

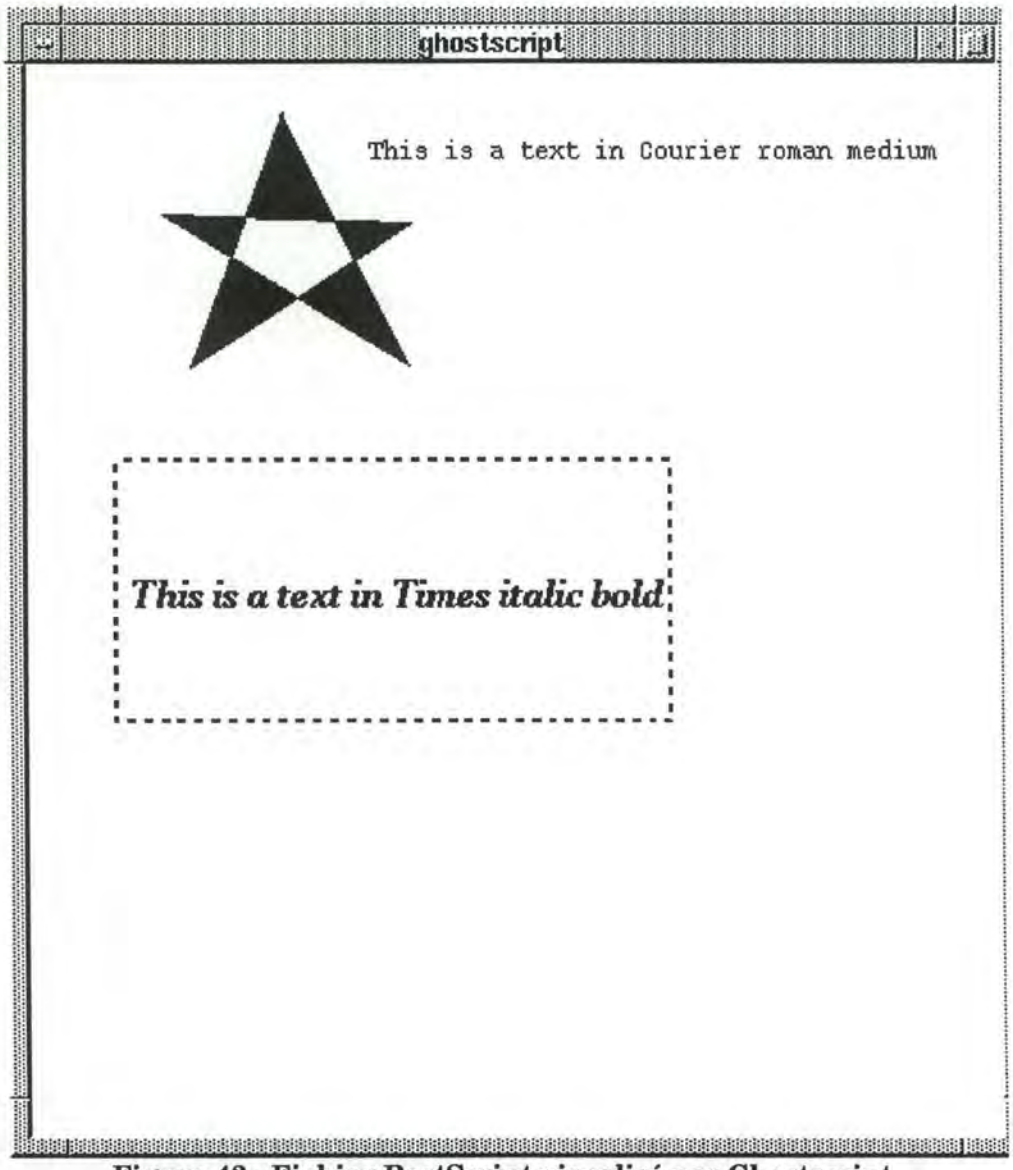

Figure 42 - Fichier PostScript visualisé par Ghostscript

6.5. *Un serveur X d'impression à part entière*

La deuxième partie consistait à transformer un serveur X d'affichage muni de notre extension en un serveur X d'impression indépendant de tout système d'affichage. Pour ce faire, il avait été décidé que nous débarasserions un système normal de base de tout code superflu, c'est-à-dire de tout code accédant aux périphériques d'entrée (clavier et souris) et à l'écran.

Nous l'avons vu, le répertoire *X/mit/server/ddx* contient les fichiers sources qui agissent directement sur le matériel. Par exemple, le répertoire X/*mit/server/ddx/sun* contient le code de gestion de la souris, du clavier et de l'écran d'une station de travail SUN. Lors de la compilation du serveur X, seul un de ces répertoires est pris en compte, en fonction du matériel sur lequel va s'exécuter le serveur. Les fonctions définies dans ce répertoire et appelées à partir d'autres fichiers (par exemple, des fichiers du répertoire *X/mit/server/dix*) doivent donc être standardisées. Par exemple, les répertoires *X/mit/server/ddx/sun* et *X/mit/server/ddx/ibm* doivent tous deux contenir la fonction **InitInput()**. Dans le cas de notre serveur X d'impression, bien qu'il ne faille pas gérer de souris, ni de clavier, il faut que ces fonctions soient présentes. Dès lors, nous avons effectué une copie des fichiers du répertoire *X/mit/server/ddx/sun* vers le répertoire *X/mit/server/ddx/printer*. Nous avons débarassé ces fichiers de tout code accédant aux différents périphériques et nous avons fait en sorte que les structures de données du serveur contiennent des valeurs plausibles.

Ainsi, la fonction obligatoire **InitInput()** doit appeler la fonction **AddInputDevice()** deux fois, une fois pour le clavier, une fois pour la souris, en passant en paramètre l'adresse de la fonction gérant le périphérique. Nous avons donc déclaré deux fonctions, **printerMouseProc()** et **printerKbdProc()** qui ne font rien.

C'est également à ce niveau que nous avons ajouté le code permettant de prendre en compte les deux paramètres supplémentaires de lancement du serveur d'impression, *-log* et *-phy*.

Une fonction supplémentaire a été ajoutée à la librairie cliente, mais elle ne donne pas lieu à l'envoi d'une requête X au serveur; elle est locale à l'application cliente. Il s'agit d'une fonction retournant une liste des serveurs X d'impression (et donc des imprimantes) accessibles à l'application:

ServersListPtr list = XMPDocumentListServers(int *num)

num: nombre de serveurs X d'impression accessibles
list: liste des serveurs X d'impression accessibles

Cette fonction consulte le fichier *printermap* qui met en relation un nom logique et le nom d'un serveur d'impression. Ces deux noms ne peuvent contenir les caractères '=' et '#' utilisés respectivement pour la séparation des deux noms et pour indiquer le début de commentaires. Par exemple, une imprimante de nom logique "Imprimante qui se trouve dans le bureau au fond à droite" est gérée par le serveur de nom "rodange.crpcu.lu:2.0". Le nom logique peut-être utilisé par l'application pour présenter à l'utilisateur une liste qui lui permettra de choisir l'imprimante qu'il désire utiliser. Le nom du serveur est utilisé par l'application pour se connecter au serveur d'impression choisi. Voici un exemple de fichier *printermap*:

```
Imprimante bureau 7 = rodange.crpcu.lu:2.0     # nec
Imprimante bureau 8 = rodange.crpcu.lu:1.0     # hp
Imprimante qui se trouve dans le bureau au fond à droite =
amarna.crpcu.lu:3.0   # apple
Imprimante Namur = muguet.info.fundp.ac.be:1.0   # digital
```

Notre application de test a été adaptée en conséquence. Très peu de modifications ont été nécessaires: l'ajout d'une commande de connexion vers le serveur d'impression, le changement d'un paramètre (*display*) des fonctions concernant le MPDocument et l'affichage d'une boîte de dialogue permettant à l'utilisateur de choisir son imprimante (en fonction des données retournées par la fonction **XMPDocumentListServers()**).

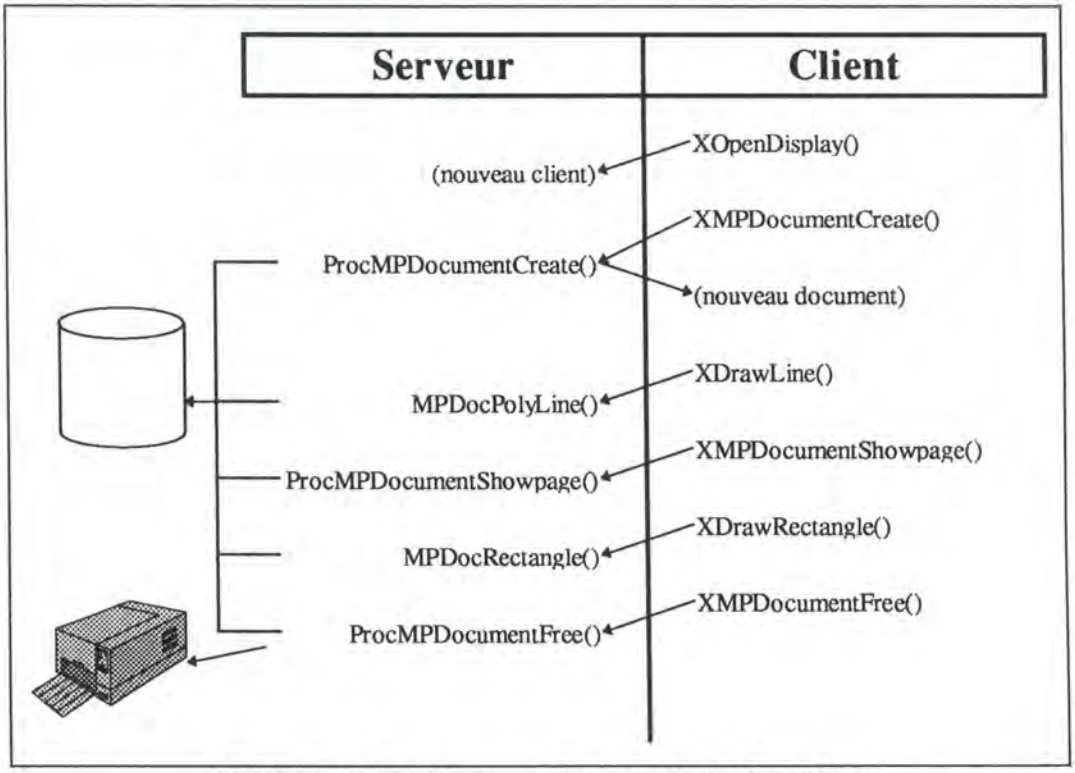

Figure 43 - La séquence des appels de fonctions

En résumé

La Figure 43 montre le déroulement d'une connexion avec un serveur X d'impression. Après ouverture d'une connexion avec le serveur X d'impression, une application cliente peut créer un document, y dessiner, passer à la page suivante et décider de l'impression du document.

6.6. *Finition du serveur*

A ce stade, le serveur X d'impression n'est pas parfait. Il lui manque plusieurs choses pour que le drawable MPDocument soit totalement comparable aux deux autres drawables que sont la fenêtre et la pixmap et que le serveur X d'impression soit parfaitement semblable à un serveur X d'affichage.

La gestion des polices devrait pouvoir se faire par l'intermédiaire des requêtes X et des fonctions de la Xlib: recherche d'une police, obtention de renseignements sur cette police, etc. Pour se faire, il faudrait remplacer les fonctions du serveur concernant les polices. Les nouvelles fonctions utiliseraient les renseignements contenus dans les fichiers Adobe Font Metrics (AFM).

De même, le protocole X devrait être augmenté de quelques requêtes permettant à l'application cliente d'obtenir des renseignements sur le périphérique géré par le serveur. Pour cela, les fichiers PPD pourraient être utilisés.

Il faudrait également gérer les couleurs et régler certains problèmes comme la copie d'une pixmap dans un MPDocument.

Enfin, les imprimantes pouvant envoyer de leur propre chef des événements (par exemple, "Out of paper") devraient être prises en compte par le serveur. Celui-ci pourrait alors envoyer à son tour un événement aux applications clientes intéressées.

Rappelons que certains de ces problèmes ont déjà été résolus et que d'autres sont à l'étude.

CONCLUSION

Nous espérons que l'argumentation suivie durant les cinq premiers chapitres aura permis d'établir que la solution du serveur d'impression était la mieux adaptée dans l'environnement X-Window.

Le dernier chapitre a démontré que l'implémentation d'un tel serveur était non seulement réalisable, mais également opérationnelle. En effet, ce dernier a été de nombreuses fois validé par l'application cliente de tests décrite dans le point 6.4.

Si nos arguments n'ont pas suffi à convaincre ou s'il subsiste certains doutes quant à l'utilisation du serveur d'impression, ce dernier est disponible gratuitement sur le serveur FTP du CRP-CU à l'adresse ftp.crpcu.lu.

Actuellement, le serveur d'impression a été implémenté sous X11R5 sur une station de travail SUN exécutant le système d'exploitation SunOS 4.1.3.. Une version disponible sous LINUX constitue une des perspectives d'avenir immédiates. De plus, le CRP-CU envisage à plus long terme de proposer le serveur X d'impression comme contribution à la distribution du Massachusetts Institute of Technology (MIT).

Grâce au serveur d'impression, le système X-Window dispose de fonctionnalités d'impression identiques à celles présentes dans les autres environnements graphiques tels Apple Macintosh, IBM OS/2 Presentation Manager et Microsoft Windows.

Dans le cadre du projet PAGE, cet ajout au système a permis aux concepteurs de GRAVITI de porter le module virtuel d'impression dans l'environnement OSF/Motif. La société SILIS, qui utilise les services de la boîte à outils virtuelle GRAVITI pour le développement de son traitement de texte graphique Interscript, va donc bientôt pouvoir commercialiser son produit dans l'environnement Motif. Cela permettra notamment de tester les possibilités d'impression de notre serveur face à une application bureautique professionnelle.

BIBLIOGRAPHIE

ADOBE SYSTEMS INC., *PostScript Language Reference Manual*, Addison-Wesley, 1989

ADOBE SYSTEMS INC., *PostScript Language Tutorial and Cookbook*, Addison-Wesley, 1989

ANGEBRANNDT S., DREWRY R., KARLTON Ph. and NEWMAN T., *Strategies for Porting the X v11 Sample Server*, MIT X Consortium, 1988

ANGEBRANNDT S., DREWRY R., KARLTON Ph. and NEWMAN T., *Definition of the Porting Layer for the X v11 Sample Server*, MIT X Consortium, 1988

APPLE COMPUTER INC., *Inside Macintosh Volume II*, Addison Wesley Publishing Company, Reading, 1985

BARBA F., MOUSEL P., NOGACKI G., RETTER P., *GRAVITI : une boîte à outils virtuelle graphique*, CRP-CU, Luxembourg

BARBA F., MOUSEL P., RETTER P., *Application Portability in Multiple Graphical Environments*, CRP-CU report CREDI-R-92-047, Luxembourg

BARBA F., MOUSEL P., NOGACKI G., RETTER P., *Improving Application Portability in Graphical Environments through the Use of Virtual Toolkits*, soumis au comité de lecture du 17th International Conference on Software Engineering (ICSE 17)

BRISTOL TECHNOLOGY INC., *Xprinter - The PostScript and PCL printer language library for the X Window System*, Bristol Technology, 1993

CRP-CU et OFFIS, *Interfaces Utilisateur Graphiques sous UNIX - Cours de formation au système UNIX*, Luxembourg

CRP-CU et OFFIS, *Utilisation du système UNIX - Cours de formation au système UNIX*, Luxembourg

CRP-CU et son Centre de Formation, *Réseaux de Communication Internet*, Luxembourg, 1994

DEININGER A., *An X Print Server – Bringing WYSIWYG to X*, The X Resource, n°10, 1994

DELPERDANGE Th., DEMANGE A., GUILLAUME C., MAILLET E., MOUSEL P., POLEUR M., RETTER P., *Design and Implementation of an X Print Server*, soumis au comité de lecture de The X Resource

DEMANGE A., GUILLAUME C., *Rapport de stage*, CRP-CU/MIAGE NANCY II, 1994

HOLZGANG D., *Comprendre PostScript - Niveaux 1 et 2*, SYBEX, 1992

MAILLET E., *Conception d'un serveur d'impression sous X Window*, CRP-CU, Luxembourg, 1993

MANSFIELD N., *The joy of X*, Addison-Wesley, 1992

MEINADIER J.P., *L'interface utilisateur - Pour une informatique plus conviviale,* DUNOD, 1991

MOUSEL P., NOGACKI G., RETTER P., *Maximum Abstraction as a Path Towards Portability in Multiple Graphical Environments*, CRP-CU report CREDI-R-92-009, Luxembourg

MOUSEL P., NOGACKI G., RETTER P., *Maximum Abstraction as a Path Towards Portability in Multiple Graphical Environments - Presentation at the IFIP WG2.7 Working Conference*, CRP-CU report CREDI-R-92-055

MOUSEL P. and RETTER P., *La portabilité des applications informatiques dans des environnements graphiques multiples*, CRP-CU report CREDI-A-91-004, Luxembourg (1991)

NYE A., *X Protocol Reference Manual*, O'Reilly & Associates, Sebastopol, 1990

NYE A., *Xlib Programming Manual*, O'Reilly & Associates, Sebastopol, 1990

PETZOLD C., *Programming Windows 3.1*, Microsoft Press, Redmond, 1992

RAVES W., *A PostScript X Server*, The X Resource, n°1, 1992

ZEIPPEN J.-M., *OBLOG - An Objet-Oriented, Formal, Tool-Based Software Engineering Approach*, Institut d'Informatique, FUNDP Namur, March-April 1994